pandas: powerful Python data analysis toolkit

Release 1.4.2

Wes McKinney and the Pandas Development Team

Apr 02, 2022
## CONTENTS

### 1 Getting started
1.1 Installation .......................................................... 3
1.2 Intro to pandas ...................................................... 3
1.3 Coming from... .................................................... 5
1.4 Tutorials .............................................................. 5
1.4.1 Installation ....................................................... 6
1.4.2 Package overview ............................................... 12
1.4.3 Getting started tutorials ...................................... 15
1.4.4 Comparison with other tools .................................. 61
1.4.5 Community tutorials .......................................... 147

### 2 User Guide
2.1 10 minutes to pandas .............................................. 149
2.1.1 Object creation ................................................ 149
2.1.2 Viewing data ................................................... 151
2.1.3 Selection ........................................................ 153
2.1.4 Missing data .................................................... 158
2.1.5 Operations ...................................................... 159
2.1.6 Merge ............................................................ 161
2.1.7 Grouping ........................................................ 163
2.1.8 Reshaping ....................................................... 164
2.1.9 Time series ..................................................... 167
2.1.10 Categoricals ................................................... 169
2.1.11 Plotting ......................................................... 170
2.1.12 Getting data in/out .......................................... 172
2.1.13 Gotchas ......................................................... 174
2.2 Intro to data structures ........................................... 174
2.2.1 Series ........................................................... 174
2.2.2 DataFrame ....................................................... 180
2.3 Essential basic functionality .................................... 198
2.3.1 Head and tail ................................................... 198
2.3.2 Attributes and underlying data ............................. 199
2.3.3 Accelerated operations ....................................... 201
2.3.4 Flexible binary operations .................................. 202
2.3.5 Descriptive statistics ........................................ 211
2.3.6 Function application ......................................... 220
2.3.7 Reindexing and altering labels ............................. 233
2.3.8 Iteration ........................................................ 243
2.3.9 .dt accessor ..................................................... 247
2.3.10 Vectorized string methods ................................. 251
2.6 MultiIndex / advanced indexing ............................................. 470
  2.6.1 Hierarchical indexing (MultiIndex) ................................... 470
  2.6.2 Advanced indexing with hierarchical index ......................... 477
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.20</td>
<td>Time series / date functionality</td>
<td>839</td>
</tr>
<tr>
<td>2.20.1</td>
<td>Overview</td>
<td>841</td>
</tr>
<tr>
<td>2.20.2</td>
<td>Timestamps vs. time spans</td>
<td>842</td>
</tr>
<tr>
<td>2.20.3</td>
<td>Converting to timestamps</td>
<td>844</td>
</tr>
<tr>
<td>2.20.4</td>
<td>Generating ranges of timestamps</td>
<td>848</td>
</tr>
<tr>
<td>2.20.5</td>
<td>Timestamp limitations</td>
<td>851</td>
</tr>
<tr>
<td>2.20.6</td>
<td>Indexing</td>
<td>852</td>
</tr>
<tr>
<td>2.20.7</td>
<td>Time/date components</td>
<td>861</td>
</tr>
<tr>
<td>2.20.8</td>
<td>DateOffset objects</td>
<td>862</td>
</tr>
<tr>
<td>2.20.9</td>
<td>Time series-related instance methods</td>
<td>878</td>
</tr>
<tr>
<td>2.20.10</td>
<td>Resampling</td>
<td>880</td>
</tr>
<tr>
<td>2.20.11</td>
<td>Time span representation</td>
<td>891</td>
</tr>
<tr>
<td>2.20.12</td>
<td>Converting between representations</td>
<td>898</td>
</tr>
<tr>
<td>2.20.13</td>
<td>Representing out-of-bounds spans</td>
<td>899</td>
</tr>
<tr>
<td>2.20.14</td>
<td>Time zone handling</td>
<td>900</td>
</tr>
<tr>
<td>2.21</td>
<td>Time deltas</td>
<td>909</td>
</tr>
<tr>
<td>2.21.1</td>
<td>Parsing</td>
<td>909</td>
</tr>
<tr>
<td>2.21.2</td>
<td>Operations</td>
<td>911</td>
</tr>
<tr>
<td>2.21.3</td>
<td>Reductions</td>
<td>915</td>
</tr>
<tr>
<td>2.21.4</td>
<td>Frequency conversion</td>
<td>916</td>
</tr>
<tr>
<td>2.21.5</td>
<td>Attributes</td>
<td>918</td>
</tr>
<tr>
<td>2.21.6</td>
<td>TimedeltaIndex</td>
<td>920</td>
</tr>
<tr>
<td>2.21.7</td>
<td>Resampling</td>
<td>924</td>
</tr>
<tr>
<td>2.22</td>
<td>Options and settings</td>
<td>924</td>
</tr>
<tr>
<td>2.22.1</td>
<td>Overview</td>
<td>924</td>
</tr>
<tr>
<td>2.22.2</td>
<td>Available options</td>
<td>926</td>
</tr>
<tr>
<td>2.22.3</td>
<td>Getting and setting options</td>
<td>933</td>
</tr>
<tr>
<td>2.22.4</td>
<td>Setting startup options in Python/IPython environment</td>
<td>934</td>
</tr>
<tr>
<td>2.22.5</td>
<td>Frequently used options</td>
<td>934</td>
</tr>
<tr>
<td>2.22.6</td>
<td>Number formatting</td>
<td>941</td>
</tr>
<tr>
<td>2.22.7</td>
<td>Unicode formatting</td>
<td>942</td>
</tr>
<tr>
<td>2.22.8</td>
<td>Table schema display</td>
<td>943</td>
</tr>
<tr>
<td>2.23</td>
<td>Enhancing performance</td>
<td>943</td>
</tr>
<tr>
<td>2.23.1</td>
<td>Cython (writing C extensions for pandas)</td>
<td>944</td>
</tr>
<tr>
<td>2.23.2</td>
<td>Numba (JIT compilation)</td>
<td>949</td>
</tr>
<tr>
<td>2.23.3</td>
<td>Expression evaluation via <code>eval()</code></td>
<td>951</td>
</tr>
<tr>
<td>2.24</td>
<td>Scaling to large datasets</td>
<td>960</td>
</tr>
<tr>
<td>2.24.1</td>
<td>Load less data</td>
<td>960</td>
</tr>
<tr>
<td>2.24.2</td>
<td>Use efficient datatypes</td>
<td>962</td>
</tr>
<tr>
<td>2.24.3</td>
<td>Use chunking</td>
<td>963</td>
</tr>
<tr>
<td>2.24.4</td>
<td>Use other libraries</td>
<td>965</td>
</tr>
<tr>
<td>2.25</td>
<td>Sparse data structures</td>
<td>969</td>
</tr>
<tr>
<td>2.25.1</td>
<td>SparseArray</td>
<td>971</td>
</tr>
<tr>
<td>2.25.2</td>
<td>SparseDtype</td>
<td>971</td>
</tr>
<tr>
<td>2.25.3</td>
<td>Sparse accessor</td>
<td>972</td>
</tr>
<tr>
<td>2.25.4</td>
<td>Sparse calculation</td>
<td>972</td>
</tr>
<tr>
<td>2.25.5</td>
<td>Migrating</td>
<td>973</td>
</tr>
</tbody>
</table>
3 API reference

3.1 Input/output

3.1.1 Pickling
3.1.2 Flat file
3.1.3 Clipboard
3.1.4 Excel
3.1.5 JSON
3.1.6 HTML
3.1.7 XML
3.1.8 Latex
3.1.9 HDFStore: PyTables (HDF5)
3.1.10 Feather
3.1.11 Parquet
3.1.12 ORC
3.1.13 SAS
3.1.14 SPSS
3.1.15 SQL
3.1.16 Google BigQuery
3.1.17 STATA

3.2 General functions

3.2.1 Data manipulations
3.2.2 Top-level missing data
3.2.3 Top-level dealing with numeric data
3.2.4 Top-level dealing with datetimelike data
3.2.5 Top-level dealing with Interval data
3.2.6 Top-level evaluation
3.2.7 Hashing
3.2.8 Testing

3.3 Series

3.3.1 Constructor
3.3.2 Attributes
3.3.3 Conversion ................................................................. 1446
3.3.4 Indexing, iteration ..................................................... 1447
3.3.5 Binary operator functions ......................................... 1449
3.3.6 Function application, GroupBy & window ...................... 1450
3.3.7 Computations / descriptive stats ................................. 1450
3.3.8 Reindexing / selection / label manipulation .................... 1452
3.3.9 Missing data handling ................................................. 1453
3.3.10 Reshaping, sorting .................................................. 1453
3.3.11 Combining / comparing / joining / merging .................... 1453
3.3.12 Time Series-related .................................................. 1454
3.3.13 Accessors .............................................................. 1454
3.3.14 Plotting ............................................................... 1571
3.3.15 Serialization / IO / conversion ................................... 1620

3.4 DataFrame ................................................................. 1620
3.4.1 Constructor ............................................................ 1620
3.4.2 Attributes and underlying data .................................... 1620
3.4.3 Conversion ............................................................ 1963
3.4.4 Indexing, iteration ..................................................... 1964
3.4.5 Binary operator functions ......................................... 1965
3.4.6 Function application, GroupBy & window ...................... 1966
3.4.7 Computations / descriptive stats ................................. 1966
3.4.8 Reindexing / selection / label manipulation .................... 1968
3.4.9 Missing data handling ................................................. 1969
3.4.10 Reshaping, sorting, transposing .................................. 1969
3.4.11 Combining / comparing / joining / merging .................... 1970
3.4.12 Time Series-related .................................................. 1970
3.4.13 Flags ................................................................. 1970
3.4.14 Metadata ............................................................. 1971
3.4.15 Plotting ............................................................... 1971
3.4.16 Sparse accessor ....................................................... 2025
3.4.17 Serialization / IO / conversion ................................... 2027

3.5 pandas arrays, scalars, and data types ............................ 2027
3.5.1 pandas.array ........................................................... 2028
3.5.2 Datetime data .......................................................... 2031
3.5.3 Timedelta data .......................................................... 2064
3.5.4 Timespan data .......................................................... 2074
3.5.5 Period ................................................................. 2074
3.5.6 Interval data ............................................................ 2092
3.5.7 Nullable integer ......................................................... 2106
3.5.8 Categorical data ......................................................... 2111
3.5.9 Sparse data ............................................................. 2117
3.5.10 Text data .............................................................. 2119
3.5.11 Boolean data with missing values ............................... 2122

3.6 Index objects ............................................................. 2124
3.6.1 Index ................................................................. 2124
3.6.2 Numeric Index ........................................................ 2192
3.6.3 CategoricalIndex ...................................................... 2196
3.6.4 IntervalIndex .......................................................... 2206
3.6.5 MultiIndex ............................................................. 2218
3.6.6 DatetimeIndex ........................................................ 2218
3.6.7 TimedeltaIndex ........................................................ 2272
3.6.8 PeriodIndex ............................................................ 2282

3.7 Date offsets .............................................................. 2290
3.7.1 DateOffset ............................................................. 2290
3.7.2 BusinessDay ........................................ 2296
3.7.3 BusinessHour ..................................... 2303
3.7.4 CustomBusinessDay ................................. 2309
3.7.5 CustomBusinessHour ................................. 2316
3.7.6 MonthEnd . ........................................ 2322
3.7.7 MonthBegin ........................................ 2327
3.7.8 BusinessMonthEnd ................................ 2332
3.7.9 BusinessMonthBegin ............................... 2338
3.7.10 CustomBusinessMonthEnd ....................... 2343
3.7.11 CustomBusinessMonthBegin ..................... 2350
3.7.12 SemiMonthEnd .................................. 2357
3.7.13 SemiMonthBegin ................................ 2362
3.7.14 Week .............................................. 2368
3.7.15 WeekOfMonth ..................................... 2373
3.7.16 LastWeekOfMonth ................................ 2379
3.7.17 BQuarterEnd ..................................... 2385
3.7.18 BQuarterBegin .................................... 2391
3.7.19 QuarterEnd ........................................ 2397
3.7.20 QuarterBegin ..................................... 2402
3.7.21 BYearEnd .......................................... 2408
3.7.22 BYearBegin ........................................ 2414
3.7.23 YearEnd ........................................... 2419
3.7.24 YearBegin ......................................... 2424
3.7.25 FY5253 ............................................. 2429
3.7.26 FY5253Quarter .................................... 2436
3.7.27 Easter .............................................. 2443
3.7.28 Tick .................................................. 2449
3.7.29 Day .................................................. 2454
3.7.30 Hour ............................................... 2460
3.7.31 Minute ............................................. 2465
3.7.32 Second ............................................. 2471
3.7.33 Milli ............................................... 2476
3.7.34 Micro ............................................... 2482
3.7.35 Nano ............................................... 2487
3.8 Frequencies .......................................... 2493
3.8.1 pandas.tseries.frequencies.to_offset .......... 2493
3.9 Window ............................................... 2494
3.9.1 Rolling window functions ....................... 2494
3.9.2 Weighted window functions ..................... 2512
3.9.3 Expanding window functions .................... 2514
3.9.4 Exponentially-weighted window functions .... 2528
3.9.5 Window indexer ................................... 2532
3.10 GroupBy ............................................. 2535
3.10.1 Indexing, iteration ............................... 2535
3.10.2 Function application ............................. 2540
3.10.3 Computations / descriptive stats ............... 2552
3.11 Resampling ......................................... 2608
3.11.1 Indexing, iteration ............................... 2608
3.11.2 Function application ............................. 2611
3.11.3 Upsampling ...................................... 2616
3.11.4 Computations / descriptive stats ............... 2628
3.12 Style ................................................. 2634
3.12.1 Styler constructor ............................... 2634
3.12.2 Styler properties ................................ 2677
4 Development

4.1 Contributing to pandas
   4.1.1 Where to start?
   4.1.2 Bug reports and enhancement requests
   4.1.3 Working with the code
   4.1.4 Contributing your changes to pandas
   4.1.5 Tips for a successful pull request

4.2 Creating a development environment
   4.2.1 Creating an environment using Docker
   4.2.2 Creating an environment without Docker

4.3 Contributing to the documentation
   4.3.1 About the pandas documentation
   4.3.2 Updating a pandas docstring
   4.3.3 How to build the pandas documentation
   4.3.4 Previewing changes

4.4 Contributing to the code base
   4.4.1 Code standards
   4.4.2 Pre-commit
   4.4.3 Optional dependencies
   4.4.4 Type hints
   4.4.5 Testing with continuous integration
   4.4.6 Test-driven development/code writing

3.12.3 Style application .................................................. 2678
3.12.4 Built-in styles .................................................. 2679
3.12.5 Style export and import ........................................ 2679

3.13 Plotting ................................................................. 2679
   3.13.1 pandas.plotting.andrews_curves ................................ 2680
   3.13.2 pandas.plotting.pandas.plot.autocorrelation_plot .............. 2680
   3.13.3 pandas.plotting.pandas.plot.bootstack_plot .................. 2682
   3.13.4 pandas.plotting.pandas.plot.boxplot ........................ 2684
   3.13.5 pandas.plotting.pandas.plot.deregister_matplotlib_converters 2690
   3.13.6 pandas.plotting.pandas.plot.lag_plot ........................ 2690
   3.13.7 pandas.plotting.pandas.plot.parallel_coordinates ............ 2693
   3.13.8 pandas.plotting.pandas.plot.plot .................................. 2693
   3.13.9 pandas.plotting.pandas.plot.radviz .......................... 2695
   3.13.10 pandas.plotting.pandas.plot.register_matplotlib_converters 2696
   3.13.11 pandas.plotting.pandas.plot.scatter_matrix ................ 2697
   3.13.12 pandas.plotting.pandas.plot.table ........................ 2699

3.14 General utility functions ........................................ 2699
   3.14.1 Working with options ........................................ 2699
   3.14.2 Testing functions ........................................... 2718
   3.14.3 Exceptions and warnings .................................... 2723
   3.14.4 Data types related functionality ............................ 2728
   3.14.5 Bug report function ......................................... 2756

3.15 Extensions ............................................................ 2757
   3.15.1 pandas.api.extensions.pandas.api.extensions.register_extension_dtype 2757
   3.15.2 pandas.api.extensions.pandas.api.extensions.register_dataframe_accessor 2757
   3.15.3 pandas.api.extensions.pandas.api.extensions.register_series_accessor 2759
   3.15.4 pandas.api.extensions.pandas.api.extensions.register_index_accessor 2760
   3.15.5 pandas.api.extensions.pandas.api.extensions.register_extension_dtype 2761
   3.15.6 pandas.api.extensions.pandas.api.extensions.register_extensionarray 2765
   3.15.7 pandas.api.extensions.pandas.api.extensions.register_extensionarray 2779
   3.15.8 pandas.api.extensions.pandas.api.extensions.register_extensionarray 2779

ix
4.4.7 Running the test suite .......................................................... 2825
4.4.8 Running the performance test suite .................................... 2826
4.4.9 Documenting your code ...................................................... 2827
4.5 pandas code style guide ......................................................... 2827
4.5.1 Patterns ................................................................. 2828
4.5.2 Testing ................................................................. 2828
4.5.3 Miscellaneous ......................................................... 2828
4.6 pandas maintenance ............................................................ 2829
4.6.1 Roles ................................................................. 2829
4.6.2 Tasks ................................................................. 2829
4.6.3 Issue triage ............................................................ 2830
4.6.4 Closing issues .......................................................... 2830
4.6.5 Reviewing pull requests ................................................. 2831
4.6.6 Backporting ............................................................ 2831
4.6.7 Cleaning up old issues .................................................. 2831
4.6.8 Cleaning up old pull requests ......................................... 2831
4.6.9 Becoming a pandas maintainer ....................................... 2832
4.6.10 Merging pull requests ................................................ 2832
4.7 Internals ................................................................. 2832
4.7.1 Indexing ............................................................... 2832
4.7.2 Subclassing pandas data structures .................................. 2834
4.8 Test organization ............................................................. 2834
4.9 Debugging C extensions ...................................................... 2837
4.9.1 Using a debugger ....................................................... 2837
4.9.2 Checking memory leaks with valgrind ............................. 2838
4.10 Extending pandas ............................................................ 2838
4.10.1 Registering custom accessors ...................................... 2838
4.10.2 Extension types ....................................................... 2839
4.10.3 Subclassing pandas data structures ............................... 2842
4.10.4 Plotting backends ..................................................... 2845
4.11 Developer ................................................................. 2846
4.11.1 Storing pandas DataFrame objects in Apache Parquet format 2846
4.12 Policies ................................................................. 2849
4.12.1 Version policy ......................................................... 2849
4.12.2 Python support .......................................................... 2849
4.13 Roadmap ................................................................. 2849
4.13.1 Extensibility ............................................................ 2850
4.13.2 String data type ....................................................... 2850
4.13.3 Consistent missing value handling .................................. 2850
4.13.4 Apache Arrow interoperability ...................................... 2850
4.13.5 Block manager rewrite ............................................... 2851
4.13.6 Decoupling of indexing and internals ............................ 2851
4.13.7 Numba-accelerated operations ..................................... 2851
4.13.8 Performance monitoring ............................................. 2851
4.13.9 Roadmap evolution ................................................... 2852
4.13.10 Completed items ..................................................... 2852
4.14 Developer meetings ......................................................... 2853
4.14.1 Minutes ................................................................. 2853
4.14.2 Calendar ................................................................. 2853

5 Release notes ................................................................. 2855
5.1 Version 1.4 ................................................................. 2855
5.1.1 What’s new in 1.4.2 (April 2, 2022) ................................. 2855
5.1.2 What’s new in 1.4.1 (February 12, 2022) .......................... 2856
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12.1</td>
<td>December 24, 2016</td>
</tr>
<tr>
<td>0.12.2</td>
<td>November 3, 2016</td>
</tr>
<tr>
<td>0.12.3</td>
<td>October 2, 2016</td>
</tr>
<tr>
<td>0.13</td>
<td>November 3, 2016</td>
</tr>
<tr>
<td>0.13.1</td>
<td>May 3, 2016</td>
</tr>
<tr>
<td>0.13.2</td>
<td>March 13, 2016</td>
</tr>
<tr>
<td>0.14</td>
<td>October 2, 2016</td>
</tr>
<tr>
<td>0.14.1</td>
<td>November 21, 2015</td>
</tr>
<tr>
<td>0.14.2</td>
<td>October 9, 2015</td>
</tr>
<tr>
<td>0.15</td>
<td>November 21, 2015</td>
</tr>
<tr>
<td>0.15.1</td>
<td>June 12, 2015</td>
</tr>
<tr>
<td>0.15.2</td>
<td>May 11, 2015</td>
</tr>
<tr>
<td>0.15.3</td>
<td>March 22, 2015</td>
</tr>
<tr>
<td>0.16</td>
<td>November 9, 2014</td>
</tr>
<tr>
<td>0.16.1</td>
<td>December 12, 2014</td>
</tr>
<tr>
<td>0.16.2</td>
<td>November 9, 2014</td>
</tr>
<tr>
<td>0.16.3</td>
<td>October 18, 2014</td>
</tr>
<tr>
<td>0.17</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>0.17.1</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>0.17.2</td>
<td>May 31, 2014</td>
</tr>
<tr>
<td>0.18</td>
<td>January 3, 2014</td>
</tr>
<tr>
<td>0.18.1</td>
<td>February 3, 2014</td>
</tr>
<tr>
<td>0.18.2</td>
<td>January 3, 2014</td>
</tr>
<tr>
<td>0.19</td>
<td>July 24, 2013</td>
</tr>
<tr>
<td>0.19.1</td>
<td>July 24, 2013</td>
</tr>
<tr>
<td>0.19.2</td>
<td>April 22, 2013</td>
</tr>
<tr>
<td>0.20</td>
<td>December 13, 2011</td>
</tr>
<tr>
<td>0.20.1</td>
<td>December 13, 2011</td>
</tr>
<tr>
<td>0.21</td>
<td>October 7, 2012</td>
</tr>
<tr>
<td>0.21.1</td>
<td>October 24, 2011</td>
</tr>
<tr>
<td>0.21.2</td>
<td>October 9, 2012</td>
</tr>
<tr>
<td>0.22</td>
<td>November 14, 2012</td>
</tr>
<tr>
<td>0.22.1</td>
<td>November 14, 2012</td>
</tr>
<tr>
<td>0.22.2</td>
<td>October 7, 2012</td>
</tr>
<tr>
<td>0.23</td>
<td>July 22, 2012</td>
</tr>
<tr>
<td>0.23.1</td>
<td>July 22, 2012</td>
</tr>
<tr>
<td>0.23.2</td>
<td>June 29, 2012</td>
</tr>
<tr>
<td>0.24</td>
<td>March 16, 2012</td>
</tr>
<tr>
<td>0.24.1</td>
<td>March 16, 2012</td>
</tr>
<tr>
<td>0.24.2</td>
<td>February 29, 2012</td>
</tr>
<tr>
<td>0.24.3</td>
<td>February 9, 2012</td>
</tr>
<tr>
<td>0.24.4</td>
<td>February 9, 2012</td>
</tr>
<tr>
<td>0.25</td>
<td>December 13, 2011</td>
</tr>
<tr>
<td>0.25.1</td>
<td>December 13, 2011</td>
</tr>
<tr>
<td>0.25.2</td>
<td>November 25, 2011</td>
</tr>
<tr>
<td>0.26</td>
<td>October 24, 2011</td>
</tr>
<tr>
<td>0.26.1</td>
<td>October 24, 2011</td>
</tr>
<tr>
<td>0.27</td>
<td>September 25 - October 9, 2011</td>
</tr>
</tbody>
</table>

Bibliography 3723

Python Module Index 3725
Getting started

New to pandas? Check out the getting started guides. They contain an introduction to pandas' main concepts and links to additional tutorials.

To the getting started guides

User guide

The user guide provides in-depth information on the key concepts of pandas with useful background information and explanation.

To the user guide

API reference

The reference guide contains a detailed description of the pandas API. The reference describes how the methods work and which parameters can be used. It assumes that you have an understanding of the key concepts.

To the reference guide

Developer guide

Saw a typo in the documentation? Want to improve existing functionalities? The contributing guidelines will guide you through the process of improving pandas.

To the development guide
CHAPTER ONE

GETTING STARTED

1.1 Installation

Working with conda?
pandas is part of the Anaconda distribution and can be installed with Anaconda or Miniconda:

\texttt{conda install pandas}

Prefer pip?
pandas can be installed via pip from PyPI.

\texttt{pip install pandas}

In-depth instructions?
Installing a specific version? Installing from source? Check the advanced installation page.
Learn more

1.2 Intro to pandas

Straight to tutorial…

When working with tabular data, such as data stored in spreadsheets or databases, pandas is the right tool for you. pandas will help you to explore, clean, and process your data. In pandas, a data table is called a \texttt{DataFrame}.

To introduction tutorial
To user guide
Straight to tutorial…

pandas supports the integration with many file formats or data sources out of the box (csv, excel, sql, json, parquet,…). Importing data from each of these data sources is provided by function with the prefix \texttt{read_*}. Similarly, the \texttt{to_*} methods are used to store data.

To introduction tutorial
To user guide
Straight to tutorial…
Selecting or filtering specific rows and/or columns? Filtering the data on a condition? Methods for slicing, selecting, and extracting the data you need are available in pandas.

pandas provides plotting your data out of the box, using the power of Matplotlib. You can pick the plot type (scatter, bar, boxplot,...) corresponding to your data.

To introduction tutorial
To user guide
Straight to tutorial...

There is no need to loop over all rows of your data table to do calculations. Data manipulations on a column work elementwise. Adding a column to a DataFrame based on existing data in other columns is straightforward.

To introduction tutorial
To user guide
Straight to tutorial...

Basic statistics (mean, median, min, max, counts...) are easily calculable. These or custom aggregations can be applied on the entire data set, a sliding window of the data, or grouped by categories. The latter is also known as the split-apply-combine approach.

To introduction tutorial
To user guide
Straight to tutorial...

Change the structure of your data table in multiple ways. You can melt() your data table from wide to long/tidy form or pivot() from long to wide format. With aggregations built-in, a pivot table is created with a single command.

To introduction tutorial
To user guide
Straight to tutorial...

Multiple tables can be concatenated both column wise and row wise as database-like join/merge operations are provided to combine multiple tables of data.

To introduction tutorial
To user guide
Straight to tutorial...

pandas has great support for time series and has an extensive set of tools for working with dates, times, and time-indexed data.
Datasets do not only contain numerical data. pandas provides a wide range of functions to clean textual data and extract useful information from it.

1.3 Coming from...

Are you familiar with other software for manipulating tabular data? Learn the pandas-equivalent operations compared to software you already know:

The R programming language provides the data.frame data structure and multiple packages, such as tidyverse use and extend data.frame for convenient data handling functionalities similar to pandas.

Already familiar to SELECT, GROUP BY, JOIN, etc.? Most of these SQL manipulations do have equivalents in pandas.

The data set included in the STATA statistical software suite corresponds to the pandas DataFrame. Many of the operations known from STATA have an equivalent in pandas.

Users of Excel or other spreadsheet programs will find that many of the concepts are transferrable to pandas.

The SAS statistical software suite also provides the data set corresponding to the pandas DataFrame. Also SAS vectorized operations, filtering, string processing operations, and more have similar functions in pandas.

1.4 Tutorials

For a quick overview of pandas functionality, see 10 Minutes to pandas.

You can also reference the pandas cheat sheet for a succinct guide for manipulating data with pandas.

The community produces a wide variety of tutorials available online. Some of the material is enlisted in the community contributed Community tutorials.
1.4.1 Installation

The easiest way to install pandas is to install it as part of the Anaconda distribution, a cross platform distribution for data analysis and scientific computing. This is the recommended installation method for most users.

Instructions for installing from source, PyPI, ActivePython, various Linux distributions, or a development version are also provided.

Python version support

Officially Python 3.8, 3.9 and 3.10.

Installing pandas

Installing with Anaconda

Installing pandas and the rest of the NumPy and SciPy stack can be a little difficult for inexperienced users. The simplest way to install not only pandas, but Python and the most popular packages that make up the SciPy stack (IPython, NumPy, Matplotlib, ...) is with Anaconda, a cross-platform (Linux, macOS, Windows) Python distribution for data analytics and scientific computing.

After running the installer, the user will have access to pandas and the rest of the SciPy stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for Anaconda can be found here.

A full list of the packages available as part of the Anaconda distribution can be found here.

Another advantage to installing Anaconda is that you don’t need admin rights to install it. Anaconda can install in the user’s home directory, which makes it trivial to delete Anaconda if you decide (just delete that folder).

Installing with Miniconda

The previous section outlined how to get pandas installed as part of the Anaconda distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with Miniconda may be a better solution.

Conda is the package manager that the Anaconda distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

Miniconda allows you to create a minimal self contained Python installation, and then use the Conda command to install additional packages.

First you will need Conda to be installed and downloading and running the Miniconda will do this for you. The installer can be found here.

The next step is to create a new conda environment. A conda environment is like a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:
source activate name_of_my_env

On Windows the command is:
activate name_of_my_env

The final step required is to install pandas. This can be done with the following command:
conda install pandas

To install a specific pandas version:
conda install pandas=0.20.3

To install other packages, IPython for example:
conda install ipython

To install the full Anaconda distribution:
conda install anaconda

If you need packages that are available to pip but not conda, then install pip, and then use pip to install those packages:
conda install pip
pip install django

**Installing from PyPI**

pandas can be installed via pip from PyPI.

**Note:** You must have pip>=19.3 to install from PyPI.

pip install pandas

**Installing with ActivePython**

Installation instructions for ActivePython can be found here. Versions 2.7, 3.5 and 3.6 include pandas.

**Installing using your Linux distribution’s package manager.**

The commands in this table will install pandas for Python 3 from your distribution.
Distributor | Status | Download / Repository Link | Install method |
---|---|---|---|
Debian | stable | official Debian repository | sudo apt-get install python3-pandas |
Debian & Ubuntu | unstable (latest packages) | NeuroDebian | sudo apt-get install python3-pandas |
Ubuntu | stable | official Ubuntu repository | sudo apt-get install python3-pandas |
OpenSuse | stable | OpenSuse Repository | zypper in python3-pandas |
Fedora | stable | official Fedora repository | dnf install python3-pandas |
Centos/RHEL | stable | EPEL repository | yum install python3-pandas |

**However**, the packages in the Linux package managers are often a few versions behind, so to get the newest version of pandas, it’s recommended to install using the pip or conda methods described above.

**Handling ImportErrors**

If you encounter an ImportError, it usually means that Python couldn’t find pandas in the list of available libraries. Python internally has a list of directories it searches through, to find packages. You can obtain these directories with:

```python
import sys
sys.path
```

One way you could be encountering this error is if you have multiple Python installations on your system and you don’t have pandas installed in the Python installation you’re currently using. In Linux/Mac you can run `which python` on your terminal and it will tell you which Python installation you’re using. If it’s something like “/usr/bin/python”, you’re using the Python from the system, which is not recommended.

It is highly recommended to use conda, for quick installation and for package and dependency updates. You can find simple installation instructions for pandas in this document: [installation instructions](https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html).

**Installing from source**

See the contributing guide for complete instructions on building from the git source tree. Further, see creating a development environment if you wish to create a pandas development environment.

**Running the test suite**

pandas is equipped with an exhaustive set of unit tests, covering about 97% of the code base as of this writing. To run it on your machine to verify that everything is working (and that you have all of the dependencies, soft and hard, installed), make sure you have pytest >= 6.0 and Hypothesis >= 3.58, then run:

```bash
>>> pd.test()
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-packages\pandas
```

(continues on next page)
Dependencies

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum supported version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>1.18.5</td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.8.1</td>
</tr>
<tr>
<td>pytz</td>
<td>2020.1</td>
</tr>
</tbody>
</table>

Recommended dependencies

- **numexpr**: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups. If installed, must be Version 2.7.1 or higher.
- **bottleneck**: for accelerating certain types of `nan` evaluations. `bottleneck` uses specialized cython routines to achieve large speedups. If installed, must be Version 1.3.1 or higher.

**Note**: You are highly encouraged to install these libraries, as they provide speed improvements, especially when working with large data sets.

Optional dependencies

Pandas has many optional dependencies that are only used for specific methods. For example, `pandas.read_hdf()` requires the `pytables` package, while `DataFrame.to_markdown()` requires the `tabulate` package. If the optional dependency is not installed, pandas will raise an `ImportError` when the method requiring that dependency is called.

Visualization

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib</td>
<td>3.3.2</td>
<td>Plotting library</td>
</tr>
<tr>
<td>Jinja2</td>
<td>2.11</td>
<td>Conditional formatting with DataFrame.style</td>
</tr>
<tr>
<td>tabulate</td>
<td>0.8.7</td>
<td>Printing in Markdown-friendly format (see tabulate)</td>
</tr>
</tbody>
</table>
Computations

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SciPy</td>
<td>1.4.1</td>
<td>Miscellaneous statistical functions</td>
</tr>
<tr>
<td>numba</td>
<td>0.50.1</td>
<td>Alternative execution engine for rolling operations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(see Enhancing Performance)</td>
</tr>
<tr>
<td>xarray</td>
<td>0.15.1</td>
<td>pandas-like API for N-dimensional data</td>
</tr>
</tbody>
</table>

**Excel files**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlrd</td>
<td>2.0.1</td>
<td>Reading Excel</td>
</tr>
<tr>
<td>xlwt</td>
<td>1.3.0</td>
<td>Writing Excel</td>
</tr>
<tr>
<td>xlsxwriter</td>
<td>1.2.2</td>
<td>Writing Excel</td>
</tr>
<tr>
<td>openpyxl</td>
<td>3.0.3</td>
<td>Reading / writing for xlsx files</td>
</tr>
<tr>
<td>pyxlsb</td>
<td>1.0.6</td>
<td>Reading for xlsb files</td>
</tr>
</tbody>
</table>

**HTML**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeautifulSoup4</td>
<td>4.8.2</td>
<td>HTML parser for read_html</td>
</tr>
<tr>
<td>html5lib</td>
<td>1.1</td>
<td>HTML parser for read_html</td>
</tr>
<tr>
<td>lxml</td>
<td>4.5.0</td>
<td>HTML parser for read_html</td>
</tr>
</tbody>
</table>

One of the following combinations of libraries is needed to use the top-level `read_html()` function:

- BeautifulSoup4 and html5lib
- BeautifulSoup4 and lxml
- BeautifulSoup4 and html5lib and lxml
- Only lxml, although see HTML Table Parsing gotchas for reasons as to why you should probably not take this approach.

**Warning:**

- if you install BeautifulSoup4 you must install either lxml or html5lib or both. `read_html()` will not work with only BeautifulSoup4 installed.
- You are highly encouraged to read HTML Table Parsing gotchas. It explains issues surrounding the installation and usage of the above three libraries.
**XML**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxml</td>
<td>4.5.0</td>
<td>XML parser for read_xml and tree builder for to_xml</td>
</tr>
</tbody>
</table>

**SQL databases**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAlchemy</td>
<td>1.4.0</td>
<td>SQL support for databases other than sqlite</td>
</tr>
<tr>
<td>psycopg2</td>
<td>2.8.4</td>
<td>PostgreSQL engine for sqlalchamy</td>
</tr>
<tr>
<td>pymysql</td>
<td>0.10.1</td>
<td>MySQL engine for sqlalchamy</td>
</tr>
</tbody>
</table>

**Other data sources**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTables</td>
<td>3.6.1</td>
<td>HDF5-based reading / writing</td>
</tr>
<tr>
<td>blosc</td>
<td>1.20.1</td>
<td>Compression for HDF5</td>
</tr>
<tr>
<td>zlib</td>
<td></td>
<td>Compression for HDF5</td>
</tr>
<tr>
<td>fastparquet</td>
<td>0.4.0</td>
<td>Parquet reading / writing</td>
</tr>
<tr>
<td>pyarrow</td>
<td>1.0.1</td>
<td>Parquet, ORC, and feather reading / writing</td>
</tr>
<tr>
<td>pyreadstat</td>
<td>1.1.0</td>
<td>SPSS files (.sav) reading</td>
</tr>
</tbody>
</table>

**Warning:**

- If you want to use `read_orc()`, it is highly recommended to install pyarrow using conda. The following is a summary of the environment in which `read_orc()` can work.

<table>
<thead>
<tr>
<th>System</th>
<th>Conda</th>
<th>PyPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Successful</td>
<td>Failed(pyarrow==3.0 Successful)</td>
</tr>
<tr>
<td>macOS</td>
<td>Successful</td>
<td>Failed</td>
</tr>
<tr>
<td>Windows</td>
<td>Failed</td>
<td>Failed</td>
</tr>
</tbody>
</table>

**Access data in the cloud**

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Minimum Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastspec</td>
<td>0.7.4</td>
<td>Handling files aside from simple local and HTTP</td>
</tr>
<tr>
<td>gcsfs</td>
<td>0.6.0</td>
<td>Google Cloud Storage access</td>
</tr>
<tr>
<td>pandas-gbq</td>
<td>0.14.0</td>
<td>Google Big Query access</td>
</tr>
<tr>
<td>s3fs</td>
<td>0.4.0</td>
<td>Amazon S3 access</td>
</tr>
</tbody>
</table>
Chapter 1. Getting started

1.4.2 Package overview

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis/manipulation tool available in any language. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, Series (1-dimensional) and DataFrame (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, DataFrame provides everything that R’s data.frame provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of missing data (represented as NaN) in floating point as well as non-float data
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
• Intuitive **merging** and **joining** data sets
• Flexible **reshaping** and pivoting of data sets
• **Hierarchical** labeling of axes (possible to have multiple labels per tick)
• Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
• **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. **pandas** is the ideal tool for all of these tasks.

Some other notes

• **pandas** is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.

• **pandas** is a dependency of **statsmodels**, making it an important part of the statistical computing ecosystem in **Python**.

• **pandas** has been used extensively in production in financial applications.

### Data structures

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Series</td>
<td>1D labeled homogeneously-typed array</td>
</tr>
<tr>
<td>2</td>
<td>DataFrame</td>
<td>General 2D labeled, size-mutable tabular structure with potentially heterogeneous-typed column</td>
</tr>
</tbody>
</table>

**Why more than one data structure?**

The best way to think about the **pandas** data structures is as flexible containers for lower dimensional data. For example, **DataFrame** is a container for **Series**, and **Series** is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using the N-dimensional array (ndarrays) to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing functions; axes are considered more or less equivalent (except when C- or Fortran-contiguousness matters for performance). In **pandas**, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set, there is likely to be a “right” way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (**DataFrame**) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the **DataFrame** thus results in more readable code:

```python
for col in df.columns:
    series = df[col]
    # do something with series
```
Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to favor immutability where sensible.

Getting support

The first stop for pandas issues and ideas is the Github Issue Tracker. If you have a general question, pandas community experts can answer through Stack Overflow.

Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to all of our contributors.

If you’re interested in contributing, please visit the contributing guide.

pandas is a NumFOCUS sponsored project. This will help ensure the success of the development of pandas as a world-class open-source project and makes it possible to donate to the project.

Project governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in Project Governance documents. The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).

Development team

The list of the Core Team members and more detailed information can be found on the people’s page of the governance repo.

Institutional partners

The information about current institutional partners can be found on pandas website page.

License

BSD 3-Clause License

Copyright (c) 2008–2011, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData Development Team
All rights reserved.

Copyright (c) 2011–2021, Open source contributors.

(continues on next page)
1.4.3 Getting started tutorials

What kind of data does pandas handle?

I want to start using pandas

In [1]: import pandas as pd

To load the pandas package and start working with it, import the package. The community agreed alias for pandas is pd, so loading pandas as pd is assumed standard practice for all of the pandas documentation.

pandas data table representation

I want to store passenger data of the Titanic. For a number of passengers, I know the name (characters), age (integers) and sex (male/female) data.

In [2]: df = pd.DataFrame(
        ...:     {
        ...:         "Name": [
        ...:             "Braund, Mr. Owen Harris",
        ...:             "Allen, Mr. William Henry",
        ...:             "Bonnell, Miss. Elizabeth",
        ...:         ],
        ...:     })
...:   "Age": [22, 35, 58],
...:   "Sex": ["male", "male", "female"],
...: )
...:

In [3]: df
Out[3]:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braund, Mr. Owen Harris</td>
<td>22</td>
<td>male</td>
</tr>
<tr>
<td>Allen, Mr. William Henry</td>
<td>35</td>
<td>male</td>
</tr>
<tr>
<td>Bonnell, Miss. Elizabeth</td>
<td>58</td>
<td>female</td>
</tr>
</tbody>
</table>

To manually store data in a table, create a DataFrame. When using a Python dictionary of lists, the dictionary keys will be used as column headers and the values in each list as columns of the DataFrame.

A DataFrame is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table or the data.frame in R.

- The table has 3 columns, each of them with a column label. The column labels are respectively Name, Age and Sex.
- The column Name consists of textual data with each value a string, the column Age are numbers and the column Sex is textual data.

In spreadsheet software, the table representation of our data would look very similar:
Each column in a DataFrame is a Series

I’m just interested in working with the data in the column Age

```
In [4]: df["Age"]
Out[4]:
0  22
1  35
2  58
Name: Age, dtype: int64
```

When selecting a single column of a pandas DataFrame, the result is a pandas Series. To select the column, use the column label in between square brackets [].

Note: If you are familiar to Python dictionaries, the selection of a single column is very similar to selection of dictionary values based on the key.

You can create a Series from scratch as well:

```
In [5]: ages = pd.Series([22, 35, 58], name="Age")
In [6]: ages
Out[6]:
0   22
1   35
2   58
Name: Age, dtype: int64
```

A pandas Series has no column labels, as it is just a single column of a DataFrame. A Series does have row labels.

Do something with a DataFrame or Series

I want to know the maximum Age of the passengers

We can do this on the DataFrame by selecting the Age column and applying max():

```
In [7]: df["Age"].max()
Out[7]: 58
```

Or to the Series:

```
In [8]: ages.max()
Out[8]: 58
```

As illustrated by the max() method, you can do things with a DataFrame or Series. pandas provides a lot of functionalities, each of them a method you can apply to a DataFrame or Series. As methods are functions, do not forget to use parentheses ()..

I’m interested in some basic statistics of the numerical data of my data table
The `describe()` method provides a quick overview of the numerical data in a DataFrame. As the Name and Sex columns are textual data, these are by default not taken into account by the `describe()` method.

Many pandas operations return a DataFrame or a Series. The `describe()` method is an example of a pandas operation returning a pandas Series or a pandas DataFrame.

Check more options on `describe` in the user guide section about aggregations with `describe`.

**Note:** This is just a starting point. Similar to spreadsheet software, pandas represents data as a table with columns and rows. Apart from the representation, also the data manipulations and calculations you would do in spreadsheet software are supported by pandas. Continue reading the next tutorials to get started!

- Import the package, aka `import pandas as pd`
- A table of data is stored as a pandas DataFrame
- Each column in a DataFrame is a Series
- You can do things by applying a method to a DataFrame or Series

A more extended explanation to DataFrame and Series is provided in the introduction to data structures.

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.
How do I read and write tabular data?

I want to analyze the Titanic passenger data, available as a CSV file.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")
```

pandas provides the `read_csv()` function to read data stored as a csv file into a pandas DataFrame. pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, ...), each of them with the prefix `read_*`.

Make sure to always have a check on the data after reading in the data. When displaying a DataFrame, the first and last 5 rows will be shown by default:

```
In [3]: titanic
Out[3]:
   PassengerId  Survived  Pclass  Name       ..
     Ticket    Fare   Cabin Embarked       ..
 0       1       3        NaN        S  Braund, Mr. Owen Harris
 1       2       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ..
 2       3       1        NaN        C  Heikkinen, Miss. Laina
 3       4       1        NaN        C  Futrelle, Mrs. Jacques Heath (Lily May Peel)
 4       5       1        NaN        S  Allen, Mr. William Henry
   ...     ...     ...        ...
448     886     0       2        NaN Montvila, Rev. Juozas
449     887     1       1        NaN Graham, Miss. Margaret Edith
450     888     1       1  Johnston, Miss. Catherine Helen "Carrie"
451     888     1       1  Graham, Miss. Margaret Edith
452     889     1       1        NaN Johnston, Miss. Catherine Helen "Carrie"
453     890     1       1        NaN Dooley, Mr. Patrick
454     891     1       1        NaN Dooley, Mr. Patrick
   ...     ...     ...        ...
[891 rows x 12 columns]
```

I want to see the first 8 rows of a pandas DataFrame.

```
In [4]: titanic.head(8)
Out[4]:
   PassengerId  Survived  Pclass  Name       ..
     Ticket    Fare   Cabin Embarked       ..
 0       1       3        NaN        S  Braund, Mr. Owen Harris
 1       2       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ..
 2       3       1        NaN        C  Heikkinen, Miss. Laina
 3       4       1        NaN        C  Futrelle, Mrs. Jacques Heath (Lily May Peel)
 4       5       1        NaN        S  Allen, Mr. William Henry
 5       6       1        NaN        C  Johnston, Miss. Catherine Helen "Carrie"
 6       7       1        NaN        S  Graham, Miss. Margaret Edith
 7       8       1       1        NaN Dooley, Mr. Patrick
 8       9       1       1        NaN Dooley, Mr. Patrick
   ...     ...     ...        ...
```

(continues on next page)
To see the first N rows of a DataFrame, use the `head()` method with the required number of rows (in this case 8) as argument.

Note: Interested in the last N rows instead? pandas also provides a `tail()` method. For example, `titanic.tail(10)` will return the last 10 rows of the DataFrame.

A check on how pandas interpreted each of the column data types can be done by requesting the pandas `dtypes` attribute:

```
In [5]: titanic.dtypes
Out[5]:
PassengerId    int64
Survived       int64
Pclass         int64
Name           object
Sex            object
Age            float64
SibSp          int64
Parch          int64
Ticket         object
Fare           float64
Cabin          object
Embarked       object
dtype: object
```

For each of the columns, the used data type is enlisted. The data types in this DataFrame are integers (int64), floats (float64) and strings (object).

Note: When asking for the `dtypes`, no brackets are used! `dtypes` is an attribute of a DataFrame and Series. Attributes of DataFrame or Series do not need brackets. Attributes represent a characteristic of a DataFrame/Series, whereas a method (which requires brackets) do something with the DataFrame/Series as introduced in the first tutorial.

My colleague requested the Titanic data as a spreadsheet.

```
In [6]: titanic.to_excel("titanic.xlsx", sheet_name="passengers", index=False)
```

Whereas `read_*` functions are used to read data to pandas, the `to_*` methods are used to store data. The `to_excel()`
method stores the data as an excel file. In the example here, the sheet_name is named passengers instead of the default Sheet1. By setting index=False the row index labels are not saved in the spreadsheet.

The equivalent read function read_excel() will reload the data to a DataFrame:

```
In [7]: titanic = pd.read_excel("titanic.xlsx", sheet_name="passengers")
```

```
In [8]: titanic.head()
Out[8]:
PassengerId  Survived  Pclass  Name     ...  Ticket  Fare  Cabin  Embarked
0           1         0       3  Braund, Mr. Owen Harris          ...  A/5 21171 7.2500  NaN     S
1           2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th...  ...  PC 17599 71.2833  C85     C
2           3         1       3  Heikkinen, Miss. Laina                ...  STON/O2. 3101282 7.9250  NaN     S
3           4         1       1  Futrelle, Mrs. Jacques Heath (Lily May Peel) ...  ...  113803 53.1000  C123     S
4           5         0       3  Allen, Mr. William Henry             ...  ...  373450 8.0500  NaN     S
[5 rows x 12 columns]
```

I’m interested in a technical summary of a DataFrame

```
In [9]: titanic.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
# Column     Non-Null Count   Dtype
--- -------- -------------- -----  
   0 PassengerId  891 non-null   int64
   1 Survived    891 non-null   int64
   2 Pclass      891 non-null   int64
   3 Name        891 non-null   object
   4 Sex         891 non-null   object
   5 Age         714 non-null   float64
   6 SibSp       891 non-null   int64
   7 Parch       891 non-null   int64
   8 Ticket      891 non-null   object
   9 Fare        891 non-null   float64
  10 Cabin       204 non-null   object
  11 Embarked    889 non-null   object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

The method info() provides technical information about a DataFrame, so let’s explain the output in more detail:

- It is indeed a DataFrame.
- There are 891 entries, i.e. 891 rows.
- Each row has a row label (aka the index) with values ranging from 0 to 890.
- The table has 12 columns. Most columns have a value for each of the rows (all 891 values are non-null). Some columns do have missing values and less than 891 non-null values.
The columns Name, Sex, Cabin and Embarked consist of textual data (strings, aka object). The other columns are numerical data with some of them whole numbers (aka integer) and others are real numbers (aka float).

The kind of data (characters, integers,...) in the different columns are summarized by listing the dtypes.

The approximate amount of RAM used to hold the DataFrame is provided as well.

Getting data in to pandas from many different file formats or data sources is supported by read_* functions.

Exporting data out of pandas is provided by different to_* methods.

The head/tail/info methods and the dtypes attribute are convenient for a first check.

For a complete overview of the input and output possibilities from and to pandas, see the user guide section about reader and writer functions.

In [1]: import pandas as pd

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()

Out[3]:
PassengerId    Survived  Pclass      Name                                      ...      
0            1          0       3  Braund, Mr. Owen Harris (Florence Briggs Th...  
1            2          1       1  Cummings, Mrs. John Bradley (Florence Briggs Th...  
2            3          1       3  Heikkinen, Miss. Laina (Helen Louise)  
3            4          1       1  Futrelle, Mrs. Jacques Heath (Lily May Peel)  
4            5          0       3  Allen, Mr. William Henry                    ]

[5 rows x 12 columns]
How do I select a subset of a DataFrame?

How do I select specific columns from a DataFrame?

I’m interested in the age of the Titanic passengers.

```
In [4]: ages = titanic["Age"]
In [5]: ages.head()
Out[5]:
   0   22.0
   1   38.0
   2   26.0
   3   35.0
   4   35.0
Name: Age, dtype: float64
```

To select a single column, use square brackets [] with the column name of the column of interest.

Each column in a DataFrame is a Series. As a single column is selected, the returned object is a pandas Series. We can verify this by checking the type of the output:

```
In [6]: type(titanic["Age"])
Out[6]: pandas.core.series.Series
```

And have a look at the shape of the output:

```
In [7]: titanic["Age"].shape
Out[7]: (891,)
```

**DataFrame.shape** is an attribute (remember tutorial on reading and writing, do not use parentheses for attributes) of a pandas Series and DataFrame containing the number of rows and columns: *(nrows, ncolumns)*. A pandas Series is 1-dimensional and only the number of rows is returned.

I’m interested in the age and sex of the Titanic passengers.

```
In [8]: age-sex = titanic[['Age', 'Sex']]
In [9]: age-sex.head()
Out[9]:
   Age   Sex
   0 22.0 male
   1 38.0 female
   2 26.0 female
   3 35.0 female
   4 35.0 male
```

To select multiple columns, use a list of column names within the selection brackets [].

**Note:** The inner square brackets define a Python list with column names, whereas the outer brackets are used to select the data from a pandas DataFrame as seen in the previous example.

The returned data type is a pandas DataFrame:
In [10]: type(titanic["Age", "Sex"])
Out[10]: pandas.core.frame.DataFrame

In [11]: titanic["Age", "Sex"].shape
Out[11]: (891, 2)

The selection returned a DataFrame with 891 rows and 2 columns. Remember, a DataFrame is 2-dimensional with both a row and column dimension.

For basic information on indexing, see the user guide section on indexing and selecting data.

**How do I filter specific rows from a DataFrame?**

I’m interested in the passengers older than 35 years.

In [12]: above_35 = titanic[titanic["Age"] > 35]

In [13]: above_35.head()
Out[13]:
```
PassengerId Survived Pclass Name  ⦁ Sex ... Parch Ticket Fare Cabin Embarked
― Sex ... Parch Ticket Fare Cabin Embarked  Name ⦁
1 2 1 1 Cumings, Mrs. John Bradley (Florence Briggs Th... ⦁ female ... 0 PC 17599 71.2833 C85  C
6 7 0 1 McCarthy, Mr. Timothy J ⦁ male ... 0 17463 51.8625 E46  S
11 12 1 1 Bonnell, Miss. Elizabeth ⦁ female ... 0 113783 26.5500 C103  S
13 14 0 3 Andersson, Mr. Anders Johan ⦁ male ... 5 347082 31.2750 NaN  S
15 16 1 2 Hewlett, Mrs. (Mary D Kingcome) ⦁ female ... 0 248706 16.0000 NaN  S
[5 rows x 12 columns]
```

To select rows based on a conditional expression, use a condition inside the selection brackets [].

The condition inside the selection brackets `titanic["Age"] > 35` checks for which rows the `Age` column has a value larger than 35:

In [14]: titanic["Age"] > 35
Out[14]:
```
0   False
1    True
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
10   False
11   False
```

(continues on next page)
The output of the conditional expression (>, but also ==, !=, <, <=,... would work) is actually a pandas Series of boolean values (either True or False) with the same number of rows as the original DataFrame. Such a Series of boolean values can be used to filter the DataFrame by putting it in between the selection brackets []. Only rows for which the value is True will be selected.

We know from before that the original Titanic DataFrame consists of 891 rows. Let's have a look at the number of rows which satisfy the condition by checking the shape attribute of the resulting DataFrame above_35:

\[
\text{In [15]: } \text{above}_35.\text{shape}
\]
\[
\text{Out[15]: } (217, 12)
\]

I'm interested in the Titanic passengers from cabin class 2 and 3.

\[
\text{In [16]: } \text{class}_23 = \text{titanic[}\text{titanic["Pclass"].isin([2, 3])]}
\]

\[
\text{In [17]: } \text{class}_23.\text{head()}
\]
\[
\text{Out[17]: }
\begin{align*}
\text{PassengerId} & \quad \text{Survived} & \quad \text{Pclass} & \quad \text{Name} & \quad \text{Sex} & \quad \text{Age} & \quad \text{SibSp} & \quad \text{Parch} \quad \text{Ticket} & \quad \text{Fare} & \quad \text{Cabin} & \quad \text{Embarked} \\
0 & 1 & 0 & 3 & \text{Braund, Mr. Owen Harris} & \text{male} & 22.0 & 1 & 0 & \text{A/5 21171} & 7.2500 & \text{NaN} & \text{S} \\
2 & 3 & 1 & 1 & \text{Heikkinen, Miss. Laina} & \text{female} & 26.0 & 0 & 0 & \text{STON/O2. 3101282} & 7.9250 & \text{NaN} & \text{S} \\
4 & 5 & 0 & 3 & \text{Allen, Mr. William Henry} & \text{male} & 35.0 & 0 & 0 & 373450 & 8.0500 & \text{NaN} & \text{S} \\
5 & 6 & 0 & 3 & \text{Moran, Mr. James} & \text{male} & \text{NaN} & 0 & 0 & 330877 & 8.4583 & \text{NaN} & \text{Q} \\
7 & 8 & 0 & 3 & \text{Palsson, Master. Gosta Leonard} & \text{male} & 2.0 & 3 & 0 & 349909 & 21.0750 & \text{NaN} & \text{S}
\end{align*}
\]

Similar to the conditional expression, the isin() conditional function returns a True for each row the values are in the provided list. To filter the rows based on such a function, use the conditional function inside the selection brackets []. In this case, the condition inside the selection brackets titanic["Pclass"].isin([2, 3]) checks for which rows the Pclass column is either 2 or 3.

The above is equivalent to filtering by rows for which the class is either 2 or 3 and combining the two statements with an | (or) operator:

\[
\text{In [18]: } \text{class}_23 = \text{titanic[}(\text{titanic["Pclass"] == 2}) \text{ | (titanic["Pclass"] == 3)}\text{]}
\]

\[
\text{In [19]: } \text{class}_23.\text{head()}
\]
\[
\text{Out[19]: }
\begin{align*}
\text{PassengerId} & \quad \text{Survived} & \quad \text{Pclass} & \quad \text{Name} & \quad \text{Sex} & \quad \text{Age} & \quad \text{SibSp} & \quad \text{Parch} \quad \text{Ticket} & \quad \text{Fare} & \quad \text{Cabin} & \quad \text{Embarked} \\
0 & 1 & 0 & 3 & \text{Braund, Mr. Owen Harris} & \text{male} & 22.0 & 1 & 0 & \text{A/5 21171} & 7.2500 & \text{NaN} & \text{S} \\
2 & 3 & 1 & 1 & \text{Heikkinen, Miss. Laina} & \text{female} & 26.0 & 0 & 0 & \text{STON/O2. 3101282} & 7.9250 & \text{NaN} & \text{S} \\
4 & 5 & 0 & 3 & \text{Allen, Mr. William Henry} & \text{male} & 35.0 & 0 & 0 & 373450 & 8.0500 & \text{NaN} & \text{S} \\
5 & 6 & 0 & 3 & \text{Moran, Mr. James} & \text{male} & \text{NaN} & 0 & 0 & 330877 & 8.4583 & \text{NaN} & \text{Q} \\
7 & 8 & 0 & 3 & \text{Palsson, Master. Gosta Leonard} & \text{male} & 2.0 & 3 & 0 & 349909 & 21.0750 & \text{NaN} & \text{S}
\end{align*}
\]
Note: When combining multiple conditional statements, each condition must be surrounded by parentheses (). Moreover, you can not use or/and but need to use the or operator | and the and operator &.

See the dedicated section in the user guide about boolean indexing or about the isin function.

I want to work with passenger data for which the age is known.

```
In [20]: age_no_na = titanic[titanic["Age"]).notna()]

In [21]: age_no_na.head()
Out[21]:
   PassengerId  Survived  Pclass   Name ...
0           1       0         3  Braund, Mr. Owen Harris ...
1           2       1         1  Cumings, Mrs. John Bradley (Florence Briggs Th... ...
2           3       1         3    Heikkinen, Miss. Laina ...
3           3       1         3  Futrelle, Mrs. Jacques Heath (Lily May Peel) ...
4           4       1         1     Allen, Mr. William Henry ...

[5 rows x 12 columns]
```

The notna() conditional function returns a True for each row the values are not an Null value. As such, this can be combined with the selection brackets [] to filter the data table.

You might wonder what actually changed, as the first 5 lines are still the same values. One way to verify is to check if the shape has changed:

```
In [22]: age_no_na.shape
Out[22]: (714, 12)
```

For more dedicated functions on missing values, see the user guide section about handling missing data.

**How do I select specific rows and columns from a DataFrame?**

I’m interested in the names of the passengers older than 35 years.

```
In [23]: adult_names = titanic.loc[titanic["Age"] > 35, "Name"]

In [24]: adult_names.head()
Out[24]:
   Cumings, Mrs. John Bradley (Florence Briggs Th...
```

In this case, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore. The `loc/iloc` operators are required in front of the selection brackets[]. When using `loc/iloc`, the part before the comma is the rows you want, and the part after the comma is the columns you want to select.

When using the column names, row labels or a condition expression, use the `loc` operator in front of the selection brackets[]. For both the part before and after the comma, you can use a single label, a list of labels, a slice of labels, a conditional expression or a colon. Using a colon specifies you want to select all rows or columns.

I’m interested in rows 10 till 25 and columns 3 to 5.

```
In [25]: titanic.iloc[9:25, 2:5]
Out[25]:
     Pclass     Name       Sex
   --- ----- ------ ------
   9     2  Nasser, Mrs. Nicholas (Adele Achem) female
   10    3  Sandstrom, Miss. Marguerite Rut female
   11    1  Bonnell, Miss. Elizabeth female
   12    3 Saundercock, Mr. William Henry male
   13    3  Andersson, Mr. Anders Johan male
   ...   ...  ...      ...
   20    2   Fynney, Mr. Joseph J male
   21    2   Beesley, Mr. Lawrence male
   22    3    McGowan, Miss. Anna "Annie" female
   23    1     Sloper, Mr. William Thompson male
   24    3  Palsson, Miss. Torborg Danira female

[16 rows x 3 columns]
```

Again, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore. When specifically interested in certain rows and/or columns based on their position in the table, use the `iloc` operator in front of the selection brackets[].

When selecting specific rows and/or columns with `loc` or `iloc`, new values can be assigned to the selected data. For example, to assign the name `anonymous` to the first 3 elements of the third column:

```
In [26]: titanic.iloc[0:3, 3] = "anonymous"
In [27]: titanic.head()
Out[27]:
   PassengerId  Survived  Pclass  Name           ...       Cabin  Embarked
             ---       ---      ---       ====          ...       ===      ---
   0            1        0       3  anonymous  ...         NaN       S
   1            2        1       1  anonymous  ...         NaN       C
   2            3        1       3  anonymous  ...         NaN       S
   3           113803    1       1  Futrelle, Mrs. Jacques Heath (Lily May Peel) ...  C123        S
```

(continues on next page)
See the user guide section on different choices for indexing to get more insight in the usage of loc and iloc.

- When selecting subsets of data, square brackets [] are used.
- Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon.
- Select specific rows and/or columns using loc when using the row and column names
- Select specific rows and/or columns using iloc when using the positions in the table
- You can assign new values to a selection based on loc/iloc.

A full overview of indexing is provided in the user guide pages on indexing and selecting data.

```python
In [1]: import pandas as pd
In [2]: import matplotlib.pyplot as plt

For this tutorial, air quality data about $NO_2$ is used, made available by openaq and using the py-openaq package. The air_quality_no2.csv data set provides $NO_2$ values for the measurement stations FR04014, BETR801 and London Westminster in respectively Paris, Antwerp and London.

In [3]: air_quality = pd.read_csv("data/air_quality_no2.csv", index_col=0, parse_dates=True)

In [4]: air_quality.head()
Out[4]:
   station_antwerp  station_paris  station_london
datetime
2019-05-07 02:00:00      NaN        NaN          23.0
2019-05-07 03:00:00  50.5         25.0          19.0
2019-05-07 04:00:00  45.0         27.7          19.0
2019-05-07 05:00:00      NaN        50.4          16.0
2019-05-07 06:00:00      NaN        61.9          NaN

Note: The usage of the index_col and parse_dates parameters of the read_csv function to define the first (0th) column as index of the resulting DataFrame and convert the dates in the column to Timestamp objects, respectively.
How to create plots in pandas?

I want a quick visual check of the data.

In [5]: air_quality.plot()
Out[5]: <AxesSubplot:xlabel='datetime'>

With a DataFrame, pandas creates by default one line plot for each of the columns with numeric data.

I want to plot only the columns of the data table with the data from Paris.

In [6]: air_quality["station_paris"].plot()
Out[6]: <AxesSubplot:xlabel='datetime'>
To plot a specific column, use the selection method of the *subset data tutorial* in combination with the `plot()` method. Hence, the `plot()` method works on both `Series` and `DataFrame`.

I want to visually compare the $N_0_2$ values measured in London versus Paris.

```python
In [7]: air_quality.plot.scatter(x="station_london", y="station_paris", alpha=0.5)
Out[7]: <AxesSubplot:xlabel='station_london', ylabel='station_paris'>
```
Apart from the default line plot when using the `plot` function, a number of alternatives are available to plot data. Let’s use some standard Python to get an overview of the available plot methods:

```
In [8]:
    ...:    method_name
    ...:    for method_name in dir(air_quality.plot)
    ...:    if not method_name.startswith('_')
    ...:
    ...:
Out[8]:
    ['area',
     'bar',
     'barh',
     'box',
     'density',
     'hexbin',
     'hist',
     'kde',
     'line',
     'pie',
     'scatter']
```

Note: In many development environments as well as IPython and Jupyter Notebook, use the TAB button to get an
overview of the available methods, for example `air_quality.plot. + TAB.

One of the options is `DataFrame.plot.box()`, which refers to a boxplot. The box method is applicable on the air quality example data:

```python
In [9]: air_quality.plot.box()
Out[9]: <AxesSubplot:>
```

For an introduction to plots other than the default line plot, see the user guide section about supported plot styles.

I want each of the columns in a separate subplot.

```python
In [10]: axs = air_quality.plot.area(figsize=(12, 4), subplots=True)
```
Separate subplots for each of the data columns are supported by the `subplots` argument of the `plot` functions. The builtin options available in each of the pandas plot functions are worth reviewing.

Some more formatting options are explained in the user guide section on `plot formatting`.

I want to further customize, extend or save the resulting plot.

```python
In [11]: fig, axs = plt.subplots(figsize=(12, 4))

In [12]: air_quality.plot.area(ax=axs)
Out[12]: <AxesSubplot:xlabel='datetime'>

In [13]: axs.set_ylabel("NO$_2$ concentration")
Out[13]: Text(0, 0.5, 'NO$_2$ concentration')

In [14]: fig.savefig("no2_concentrations.png")
```

Each of the plot objects created by pandas is a `matplotlib` object. As Matplotlib provides plenty of options to customize plots, making the link between pandas and Matplotlib explicit enables all the power of matplotlib to the plot. This strategy is applied in the previous example:

```python
fig, axs = plt.subplots(figsize=(12, 4))

air_quality.plot.area(ax=axs)

axs.set_ylabel("NO$_2$ concentration")
```

(continues on next page)
For this tutorial, air quality data about NO₂ is used, made available by openaq and using the py-openaq package. The air_quality_no2.csv data set provides NO₂ values for the measurement stations FR04014, BETR801 and London Westminster in respectively Paris, Antwerp and London.

How to create new columns derived from existing columns?

I want to express the NO₂ concentration of the station in London in mg/m³

(If we assume temperature of 25 degrees Celsius and pressure of 1013 hPa, the conversion factor is 1.882)

To create a new column, use the [] brackets with the new column name at the left side of the assignment.

Note: The calculation of the values is done element wise. This means all values in the given column are multiplied
by the value 1.882 at once. You do not need to use a loop to iterate each of the rows!

I want to check the ratio of the values in Paris versus Antwerp and save the result in a new column

```python
In [6]: air_quality["ratio_paris_antwerp"] = (  
        ...:     air_quality["station_paris"] / air_quality["station_antwerp"]  
        ...:
        ...
)
```

```python
In [7]: air_quality.head()
Out[7]:
station_antwerp  station_paris  station_london  london_mg_per_cubic
   ratio_paris_antwerp  
datetime       
2019-05-07 02:00:00  NaN       NaN     23.0              43.286  
                        NaN       NaN     NaN                  NaN  
2019-05-07 03:00:00  50.5       25.0    19.0              35.758  
                  0.495050  0.615556  
2019-05-07 04:00:00  45.0       27.7    19.0              35.758  
                  0.615556  
2019-05-07 05:00:00  NaN       50.4     16.0              30.112  
                        NaN       NaN     NaN                  NaN  
2019-05-07 06:00:00  NaN       61.9     NaN                  NaN  
                        NaN       NaN     NaN                  NaN
```

The calculation is again element-wise, so the / is applied for the values in each row.

Also other mathematical operators (+, -, *, /) or logical operators (<, >, =, ...) work element wise. The latter was already used in the subset data tutorial to filter rows of a table using a conditional expression.

If you need more advanced logic, you can use arbitrary Python code via apply().

I want to rename the data columns to the corresponding station identifiers used by openAQ

```python
In [8]: air_quality_renamed = air_quality.rename(  
        ...:                         columns={  
        ...:                             "station_antwerp": "BETR801",  
        ...:                             "station_paris": "FR04014",  
        ...:                             "station_london": "London Westminster",  
        ...:                         }  
        ...:
        ...
)
```

```python
In [9]: air_quality_renamed.head()
Out[9]:
BETR801  FR04014 London Westminster  london_mg_per_cubic  ratio_paris_antwerp  
datetime
2019-05-07 02:00:00  NaN       NaN     23.0              43.286
                        NaN       NaN     NaN                  NaN
2019-05-07 03:00:00  50.5       25.0    19.0              35.758
                  0.495050
(continues on next page)
The `rename()` function can be used for both row labels and column labels. Provide a dictionary with the keys the current names and the values the new names to update the corresponding names.

The mapping should not be restricted to fixed names only, but can be a mapping function as well. For example, converting the column names to lowercase letters can be done using a function as well:

```python
In [10]: air_quality_renamed = air_quality_renamed.rename(columns=str.lower)

In [11]: air_quality_renamed.head()
Out[11]:
    paris_antwerp
   datetime  betr801 fr04014 london westminster london_mg_per_cubic ratio_
2019-05-07 02:00:00  NaN  NaN  23.0    43.286
2019-05-07 03:00:00  NaN    NaN  19.0    35.758
2019-05-07 04:00:00  0.495050  45.0    27.7  19.0    35.758
2019-05-07 05:00:00  NaN    NaN  50.4    16.0    30.112
2019-05-07 06:00:00  NaN    NaN  61.9    NaN     NaN
```

Details about column or row label renaming is provided in the user guide section on renaming labels.

- Create a new column by assigning the output to the DataFrame with a new column name in between the [].
- Operations are element-wise, no need to loop over rows.
- Use `rename` with a dictionary or function to rename row labels or column names.

The user guide contains a separate section on column addition and deletion.

```python
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- **PassengerId**: Id of every passenger.
- **Survived**: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- **Pclass**: There are 3 classes: Class 1, Class 2 and Class 3.
- **Name**: Name of passenger.
- **Sex**: Gender of passenger.
- **Age**: Age of passenger.
- **SibSp**: Indication that passenger have siblings and spouse.
• Parch: Whether a passenger is alone or have family.
• Ticket: Ticket number of passenger.
• Fare: Indicating the fare.
• Cabin: The cabin of passenger.
• Embarked: The embarked category.

```python
In [2]: titanic = pd.read_csv("data/titanic.csv")
In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass  Name      ...  Cabin  Embarked
0            1       0        3  Braund, Mr. Owen Harris ...  NaN      S
1            2       1        1  Cumings, Mrs. John Bradley (Florence Briggs Th...  C
2            3       1        3  Heikkinen, Miss. Laina ...  NaN      S
3            4       1        1  Futrelle, Mrs. Jacques Heath (Lily May Peel) ...  S
4            5       0        3  Allen, Mr. William Henry ...  NaN      S

[5 rows x 12 columns]
```

**How to calculate summary statistics?**

**Aggregating statistics**

What is the average age of the Titanic passengers?

```python
In [4]: titanic["Age"].mean()
Out[4]: 29.69911764705882
```

Different statistics are available and can be applied to columns with numerical data. Operations in general exclude missing data and operate across rows by default.

What is the median age and ticket fare price of the Titanic passengers?

```python
In [5]: titanic[["Age", "Fare"]).median()
Out[5]:
Age   28.0000
Fare  14.4542
dtype: float64
```

The statistic applied to multiple columns of a DataFrame (the selection of two columns return a DataFrame, see the subset data tutorial) is calculated for each numeric column.
The aggregating statistic can be calculated for multiple columns at the same time. Remember the `describe` function from first tutorial?

```python
In [6]: titanic["Age", "Fare"].describe()
Out[6]:
       Age      Fare
count 714.000000  891.000000
mean  29.699118  32.204208
std   14.526497  49.693429
min   0.420000   0.000000
25%   20.125000  7.910400
50%   28.000000  14.454200
75%   38.000000  31.000000
max   80.000000  512.329200
```

Instead of the predefined statistics, specific combinations of aggregating statistics for given columns can be defined using the `DataFrame.agg()` method:

```python
In [7]: titanic.agg(
    ...
    {
        ...
        "Age": ["min", "max", "median", "skew"],
        ...
        "Fare": ["min", "max", "median", "mean"],
        ...
    }
    ...
)
Out[7]:
       Age      Fare
      min   0.420000  0.000000
      max   80.000000 512.329200
      median 28.000000 14.454200
      skew   0.389108  NaN
      mean   NaN      32.204208
```

Details about descriptive statistics are provided in the user guide section on descriptive statistics.

### Aggregating statistics grouped by category

What is the average age for male versus female Titanic passengers?

```python
In [8]: titanic["Sex", "Age"].groupby("Sex").mean()
Out[8]:
       Age
   Sex
female  27.915709
male    30.726645
```

As our interest is the average age for each gender, a subselection on these two columns is made first: `titanic["Sex", "Age"]`. Next, the `groupby()` method is applied on the Sex column to make a group per category. The average age for each gender is calculated and returned.

Calculating a given statistic (e.g. mean age) for each category in a column (e.g. male/female in the Sex column) is a common pattern. The groupby method is used to support this type of operations. More general, this fits in the more general split-apply-combine pattern:
• **Split** the data into groups

• **Apply** a function to each group independently

• **Combine** the results into a data structure

The apply and combine steps are typically done together in pandas.

In the previous example, we explicitly selected the 2 columns first. If not, the **mean** method is applied to each column containing numerical columns:

```
In [9]: titanic.groupby("Sex").mean()
Out[9]:
       PassengerId  Survived  Pclass  Age  SibSp  Parch  Fare
Sex
female  431.028662  0.742038  2.159236 27.915709  0.694268  0.649682  44.479818
male    454.147314  0.188908  2.389948 30.726645  0.429809  0.235702  25.523893
```

It does not make much sense to get the average value of the **Pclass**. if we are only interested in the average age for each gender, the selection of columns (rectangular brackets [] as usual) is supported on the grouped data as well:

```
In [10]: titanic.groupby("Sex")["Age"].mean()
Out[10]:
       Sex
   female  27.915709
   male    30.726645
Name: Age, dtype: float64
```

**Note:** The **Pclass** column contains numerical data but actually represents 3 categories (or factors) with respectively the labels ‘1’, ‘2’ and ‘3’. Calculating statistics on these does not make much sense. Therefore, pandas provides a **Categorical** data type to handle this type of data. More information is provided in the user guide **Categorical data** section.

What is the mean ticket fare price for each of the sex and cabin class combinations?

```
In [11]: titanic.groupby(["Sex", "Pclass"])["Fare"].mean()
Out[11]:
       Sex  Pclass
   female  1   106.125798
         2   21.970121
         3   16.118810
   male    1   67.226127
         2   19.741782
         3   12.661633
Name: Fare, dtype: float64
```

Grouping can be done by multiple columns at the same time. Provide the column names as a list to the **groupby()** method.

A full description on the split-apply-combine approach is provided in the user guide section on **groupby operations**.
Count number of records by category

What is the number of passengers in each of the cabin classes?

```python
In [12]: titanic["Pclass"].value_counts()
Out[12]:
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

The `value_counts()` method counts the number of records for each category in a column. The function is a shortcut, as it is actually a `groupby` operation in combination with counting of the number of records within each group:

```python
In [13]: titanic.groupby("Pclass")["Pclass"].count()
Out[13]:
Pclass
1    216
2    184
3    491
Name: Pclass, dtype: int64
```

**Note:** Both `size` and `count` can be used in combination with `groupby`. Whereas `size` includes NaN values and just provides the number of rows (size of the table), `count` excludes the missing values. In the `value_counts` method, use the `dropna` argument to include or exclude the NaN values.

The user guide has a dedicated section on `value_counts`, see page on discretization.

- Aggregation statistics can be calculated on entire columns or rows
- `groupby` provides the power of the split-apply-combine pattern
- `value_counts` is a convenient shortcut to count the number of entries in each category of a variable

A full description on the split-apply-combine approach is provided in the user guide pages about `groupby operations`.

```python
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
• Ticket: Ticket number of passenger.
• Fare: Indicating the fare.
• Cabin: The cabin of passenger.
• Embarked: The embarked category.

In [2]: titanic = pd.read_csv("data/titanic.csv")
In [3]: titanic.head()
Out[3]:
   PassengerId   Survived  Pclass   Name           Ticket  Fare  Cabin  Embarked
0          1          0        3  Braund, Mr. Owen Harris   A/5  21171   7.25  NaN   S
1          2          1        1       Cumings, Mrs. John Bradley (Florence Briggs Th...  PC 17599  71.28  C85   C
2          3          1        3       Heikkinen, Miss. Laina  STON/O2. 3101282  7.92  NaN   S
3          4          1        1  Futrelle, Mrs. Jacques Heath (Lily May Peel) 113803  53.10 C123   S
4          5          0        3           Allen, Mr. William Henry              373450  8.05  NaN   S

[5 rows x 12 columns]

This tutorial uses air quality data about \( NO_2 \) and Particulate matter less than 2.5 micrometers, made available by openaq and using the py-openaq package. The `air_quality_long.csv` data set provides \( NO_2 \) and \( PM_{2.5} \) values for the measurement stations `FR04014`, `BETR801` and `London Westminster` in respectively Paris, Antwerp and London.

The air-quality data set has the following columns:

• city: city where the sensor is used, either Paris, Antwerp or London
• country: country where the sensor is used, either FR, BE or GB
• location: the id of the sensor, either `FR04014`, `BETR801` or `London Westminster`
• parameter: the parameter measured by the sensor, either \( NO_2 \) or Particulate matter
• value: the measured value
• unit: the unit of the measured parameter, in this case ‘\( \mu g/m^3 \)’

and the index of the DataFrame is `datetime`, the datetime of the measurement.

Note: The air-quality data is provided in a so-called long format data representation with each observation on a separate row and each variable a separate column of the data table. The long/narrow format is also known as the tidy data format.

In [4]: air_quality = pd.read_csv(  "data/air_quality_long.csv", index_col="date.utc", parse_dates=True
...: )

In [5]: air_quality.head()
How to reshape the layout of tables?

Sort table rows

I want to sort the Titanic data according to the age of the passengers.

```
In [6]: titanic.sort_values(by="Age").head()
```

```
Out[6]:  PassengerId  Survived  Pclass  Name            Sex  Age  SibSp  Parch  Ticket  Fare  Cabin Embarked
        803       1  3.0000  Thomas, Master. Assad Alexander  male  0.42  0  NaN    NaN    NaN        C
        755       1  3.0000  Hamalainen, Master. Viljo  male  0.67  1  NaN    NaN    NaN        S
       644       1  3.0000     Baclini, Miss. Eugenie  female  0.75  2  NaN    NaN    NaN        C
       469       1  3.0000     Baclini, Miss. Helene Barbara  female  0.75  2  NaN    NaN    NaN        S
       78        1  3.0000  Caldwell, Master. Alden Gates  male  0.83  0  NaN    NaN    NaN        S
```

I want to sort the Titanic data according to the cabin class and age in descending order.

```
In [7]: titanic.sort_values(by=['Pclass', 'Age'], ascending=False).head()
```

```
Out[7]:  PassengerId  Survived  Pclass  Name            Sex  Age  SibSp  Parch  Ticket  Fare  Cabin Embarked
        851       0  3.0000  Svensson, Mr. Johan  male  74.0  0  NaN  347060  7.7750 NaN        S
       116       0  3.0000    Connors, Mr. Patrick  male  70.5  0  NaN   370369  7.7500 NaN        Q
       280       0  3.0000     Duane, Mr. Frank  male  65.0  0  NaN    281   7.5000 NaN        Q
       483       0  3.0000  Turkula, Mrs. (Hedwig)  female  63.0  0  NaN    4134  9.5875 NaN        S
       326       0  3.0000  Nysveen, Mr. Johan Hansen  male  61.0  0  NaN   345364  6.2375 NaN        S
```

With `Series.sort_values()`, the rows in the table are sorted according to the defined column(s). The index will follow the row order.

More details about sorting of tables is provided in the using guide section on sorting data.
Long to wide table format

Let’s use a small subset of the air quality data set. We focus on $NO_2$ data and only use the first two measurements of each location (i.e. the head of each group). The subset of data will be called `no2_subset`

```python
# filter for no2 data only
In [8]: no2 = air_quality[air_quality["parameter"] == "no2"]

# use 2 measurements (head) for each location (groupby)
In [9]: no2_subset = no2.sort_index().groupby(["location"]).head(2)

In [10]: no2_subset
Out[10]:

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antwerpen</td>
<td>BE</td>
<td>BETR801</td>
<td>no2</td>
<td>22.5</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>24.4</td>
<td>µg/m³</td>
</tr>
<tr>
<td>London</td>
<td>GB</td>
<td>London Westminster</td>
<td>no2</td>
<td>67.0</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Antwerpen</td>
<td>BE</td>
<td>BETR801</td>
<td>no2</td>
<td>53.5</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>27.4</td>
<td>µg/m³</td>
</tr>
<tr>
<td>London</td>
<td>GB</td>
<td>London Westminster</td>
<td>no2</td>
<td>67.0</td>
<td>µg/m³</td>
</tr>
</tbody>
</table>
```

I want the values for the three stations as separate columns next to each other

```python
In [11]: no2_subset.pivot(columns="location", values="value")
Out[11]:

<table>
<thead>
<tr>
<th>location</th>
<th>BE</th>
<th>FR04014</th>
<th>London Westminster</th>
</tr>
</thead>
<tbody>
<tr>
<td>date.utc</td>
<td>2019-04-09 01:00:00+00:00</td>
<td>22.5</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>2019-04-09 02:00:00+00:00</td>
<td>53.5</td>
<td>27.4</td>
</tr>
<tr>
<td></td>
<td>2019-04-09 03:00:00+00:00</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

The `pivot()` function is purely reshaping of the data: a single value for each index/column combination is required.

As pandas support plotting of multiple columns (see plotting tutorial) out of the box, the conversion from long to wide table format enables the plotting of the different time series at the same time:

```python
In [12]: no2.head()
Out[12]:

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>20.0</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>21.8</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>26.5</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>24.9</td>
<td>µg/m³</td>
</tr>
<tr>
<td>Paris</td>
<td>FR</td>
<td>FR04014</td>
<td>no2</td>
<td>21.4</td>
<td>µg/m³</td>
</tr>
</tbody>
</table>

In [13]: no2.pivot(columns="location", values="value").plot()
Out[13]: <AxesSubplot:xlabel='date.utc'>
**Note:** When the `index` parameter is not defined, the existing index (row labels) is used.

For more information about `pivot()`, see the user guide section on *pivoting DataFrame objects*.

**Pivot table**

I want the mean concentrations for $NO_2$ and $PM_{2.5}$ in each of the stations in table form.

```python
In [14]: air_quality.pivot_table(
    ...:     values="value", index="location", columns="parameter", aggfunc="mean"
    ...: )
    ...
Out[14]:
        parameter
term | no2       | pm25
--- | -------  | -------
BETR801 | 26.950920 | 23.169492
FR04014 | 29.374284 | NaN
London Westminster | 29.740050 | 13.443568
```

In the case of `pivot()`, the data is only rearranged. When multiple values need to be aggregated (in this specific case,
the values on different time steps) `pivot_table()` can be used, providing an aggregation function (e.g. mean) on how to combine these values.

Pivot table is a well known concept in spreadsheet software. When interested in summary columns for each variable separately as well, put the `margin` parameter to `True`:

```python
In [15]: air_quality.pivot_table(
    ....:     values="value",
    ....:     index="location",
    ....:     columns="parameter",
    ....:     aggfunc="mean",
    ....:     margins=True,
    ....: )
```

```
Out[15]:

<table>
<thead>
<tr>
<th>parameter</th>
<th>no2</th>
<th>pm25</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BETR801</td>
<td>26.95092</td>
<td>23.169492</td>
<td>24.982353</td>
</tr>
<tr>
<td>FR04014</td>
<td>29.374284</td>
<td>NaN</td>
<td>29.374284</td>
</tr>
<tr>
<td>All</td>
<td>29.430316</td>
<td>14.386849</td>
<td>24.222743</td>
</tr>
</tbody>
</table>
```

For more information about `pivot_table()`, see the user guide section on `pivot tables`.

**Note:** In case you are wondering, `pivot_table()` is indeed directly linked to `groupby()`. The same result can be derived by grouping on both `parameter` and `location`:

```python
air_quality.groupby(["parameter", "location"]).mean()
```

Have a look at `groupby()` in combination with `unstack()` at the user guide section on `combining stats and groupby`.

### Wide to long format

Starting again from the wide format table created in the previous section:

```python
In [16]: no2_pivoted = no2.pivot(columns="location", values="value").reset_index()

In [17]: no2_pivoted.head()
```

```
<table>
<thead>
<tr>
<th>location</th>
<th>date.utc</th>
<th>BETR801</th>
<th>FR04014</th>
<th>London Westminster</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-04-09 01:00:00+00:00</td>
<td>22.5</td>
<td>24.4</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2019-04-09 02:00:00+00:00</td>
<td>53.5</td>
<td>27.4</td>
<td>67.0</td>
<td></td>
</tr>
<tr>
<td>2019-04-09 03:00:00+00:00</td>
<td>54.5</td>
<td>34.2</td>
<td>67.0</td>
<td></td>
</tr>
<tr>
<td>2019-04-09 04:00:00+00:00</td>
<td>34.5</td>
<td>48.5</td>
<td>41.0</td>
<td></td>
</tr>
<tr>
<td>2019-04-09 05:00:00+00:00</td>
<td>46.5</td>
<td>59.5</td>
<td>41.0</td>
<td></td>
</tr>
</tbody>
</table>
```

I want to collect all air quality $NO_2$ measurements in a single column (long format)

```python
In [18]: no_2 = no2_pivoted.melt(id_vars="date.utc")
```

(continues on next page)
The `pandas.melt()` method on a DataFrame converts the data table from wide format to long format. The column headers become the variable names in a newly created column.

The solution is the short version on how to apply `pandas.melt()`. The method will `melt` all columns NOT mentioned in `id_vars` together into two columns: A column with the column header names and a column with the values itself. The latter column gets by default the name `value`.

The `pandas.melt()` method can be defined in more detail:

```python
In [20]: no_2 = no2_pivoted.melt(  
.....:     id_vars="date.utc",  
.....:     value_vars= ["BETR801", "FR04014", "London Westminster"],  
.....:     value_name="NO_2",  
.....:     var_name="id_location",  
.....: )

In [21]: no_2.head()
```

<table>
<thead>
<tr>
<th>date.utc</th>
<th>id_location</th>
<th>NO_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-04-09</td>
<td>BETR801</td>
<td>22.5</td>
</tr>
<tr>
<td>2019-04-09</td>
<td>BETR801</td>
<td>53.5</td>
</tr>
<tr>
<td>2019-04-09</td>
<td>BETR801</td>
<td>54.5</td>
</tr>
<tr>
<td>2019-04-09</td>
<td>BETR801</td>
<td>34.5</td>
</tr>
<tr>
<td>2019-04-09</td>
<td>BETR801</td>
<td>46.5</td>
</tr>
</tbody>
</table>

The result in the same, but in more detail defined:

- `value_vars` defines explicitly which columns to `melt` together
- `value_name` provides a custom column name for the values column instead of the default column name `value`
- `var_name` provides a custom column name for the column collecting the column header names. Otherwise it takes the index name or a default variable

Hence, the arguments `value_name` and `var_name` are just user-defined names for the two generated columns. The columns to melt are defined by `id_vars` and `value_vars`.

Conversion from wide to long format with `pandas.melt()` is explained in the user guide section on reshaping by melt.

- Sorting by one or more columns is supported by `sort_values`
- The `pivot` function is purely restructuring of the data, `pivot_table` supports aggregations
- The reverse of `pivot` (long to wide format) is `melt` (wide to long format)

A full overview is available in the user guide on the pages about reshaping and pivoting.
In [1]: import pandas as pd

For this tutorial, air quality data about $NO_2$ is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_no2_long.csv` data set provides $NO_2$ values for the measurement stations FR04014, BETR801 and London Westminster in respectively Paris, Antwerp and London.

In [2]: air_quality_no2 = pd.read_csv("data/air_quality_no2_long.csv", 
...:     parse_dates=\text{True})
...:

In [3]: air_quality_no2 = air_quality_no2[[\text{"date.utc", "location", 
...:     "parameter", "value"}]]

In [4]: air_quality_no2.head()
Out[4]:

<table>
<thead>
<tr>
<th>date.utc</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-06-21 00:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>20.0</td>
</tr>
<tr>
<td>2019-06-20 23:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>21.8</td>
</tr>
<tr>
<td>2019-06-20 22:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>26.5</td>
</tr>
<tr>
<td>2019-06-20 21:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>24.9</td>
</tr>
<tr>
<td>2019-06-20 20:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>21.4</td>
</tr>
</tbody>
</table>

For this tutorial, air quality data about Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_pm25_long.csv` data set provides $PM_{25}$ values for the measurement stations FR04014, BETR801 and London Westminster in respectively Paris, Antwerp and London.

In [5]: air_quality_pm25 = pd.read_csv("data/air_quality_pm25_long.csv", 
...:     parse_dates=\text{True})
...:

In [6]: air_quality_pm25 = air_quality_pm25[[\text{"date.utc", "location", 
...:     "parameter", "value"}]]

In [7]: air_quality_pm25.head()
Out[7]:

<table>
<thead>
<tr>
<th>date.utc</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-06-18 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.0</td>
</tr>
<tr>
<td>2019-06-17 08:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>6.5</td>
</tr>
<tr>
<td>2019-06-17 07:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.5</td>
</tr>
<tr>
<td>2019-06-17 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>16.0</td>
</tr>
<tr>
<td>2019-06-17 05:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>7.5</td>
</tr>
</tbody>
</table>
How to combine data from multiple tables?

Concatenating objects

I want to combine the measurements of \( NO_2 \) and \( PM_{2.5} \), two tables with a similar structure, in a single table.

```
In [8]: air_quality = pd.concat([air_quality_pm25, air_quality_no2], axis=0)
```

```
In [9]: air_quality.head()

Out[9]:

<table>
<thead>
<tr>
<th>date.utc</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-06-18 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.0</td>
</tr>
<tr>
<td>2019-06-17 08:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>6.5</td>
</tr>
<tr>
<td>2019-06-17 07:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.5</td>
</tr>
<tr>
<td>2019-06-17 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>16.0</td>
</tr>
<tr>
<td>2019-06-17 05:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>7.5</td>
</tr>
</tbody>
</table>
```

The `concat()` function performs concatenation operations of multiple tables along one of the axis (row-wise or column-wise).

By default concatenation is along axis 0, so the resulting table combines the rows of the input tables. Let’s check the shape of the original and the concatenated tables to verify the operation:

```
In [10]: print('Shape of the ``air_quality_pm25`` table: ', air_quality_pm25.shape)
Shape of the `air_quality_pm25` table:  (1110, 4)

In [11]: print('Shape of the `air_quality_no2` table: ', air_quality_no2.shape)
Shape of the `air_quality_no2` table:  (2068, 4)

In [12]: print('Shape of the resulting `air_quality` table: ', air_quality.shape)
Shape of the resulting `air_quality` table:  (3178, 4)
```

Hence, the resulting table has 3178 = 1110 + 2068 rows.

**Note:** The `axis` argument will return in a number of pandas methods that can be applied along an axis. A DataFrame has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1). Most operations like concatenation or summary statistics are by default across rows (axis 0), but can be applied across columns as well.

Sorting the table on the datetime information illustrates also the combination of both tables, with the parameter column defining the origin of the table (either no2 from table `air_quality_no2` or pm25 from table `air_quality_pm25`):

```
In [13]: air_quality = air_quality.sort_values("date.utc")
```

```
In [14]: air_quality.head()

Out[14]:

<table>
<thead>
<tr>
<th>date.utc</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2067 2019-05-07 01:00:00+00:00</td>
<td>London Westminster</td>
<td>no2</td>
<td>23.0</td>
</tr>
<tr>
<td>1003 2019-05-07 01:00:00+00:00</td>
<td>FR04014</td>
<td>no2</td>
<td>25.0</td>
</tr>
<tr>
<td>100 2019-05-07 01:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>12.5</td>
</tr>
</tbody>
</table>
```

(continues on next page)
In this specific example, the `parameter` column provided by the data ensures that each of the original tables can be identified. This is not always the case. the `concat` function provides a convenient solution with the `keys` argument, adding an additional (hierarchical) row index. For example:

```python
In [15]: air_quality_ = pd.concat([air_quality_pm25, air_quality_no2], keys=['PM25', 'NO2'])
```

```python
In [16]: air_quality_.head()
Out[16]:
```

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>date.utc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2019-06-18 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2019-06-17 08:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2019-06-17 07:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>18.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2019-06-17 06:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>16.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2019-06-17 05:00:00+00:00</td>
<td>BETR801</td>
<td>pm25</td>
<td>7.5</td>
<td></td>
</tr>
</tbody>
</table>
```

**Note:** The existence of multiple row/column indices at the same time has not been mentioned within these tutorials. Hierarchical indexing or MultiIndex is an advanced and powerful pandas feature to analyze higher dimensional data. Multi-indexing is out of scope for this pandas introduction. For the moment, remember that the function `reset_index` can be used to convert any level of an index to a column, e.g. `air_quality.reset_index(level=0)`.

Feel free to dive into the world of multi-indexing at the user guide section on advanced indexing.

More options on table concatenation (row and column wise) and how `concat` can be used to define the logic (union or intersection) of the indexes on the other axes is provided at the section on object concatenation.

### Join tables using a common identifier

Add the station coordinates, provided by the stations metadata table, to the corresponding rows in the measurements table.

**Warning:** The air quality measurement station coordinates are stored in a data file `air_quality_stations.csv`, downloaded using the py-openaq package.

```python
In [17]: stations_coord = pd.read_csv("data/air_quality_stations.csv")
```

```python
In [18]: stations_coord.head()
Out[18]:
```

```
<table>
<thead>
<tr>
<th>location</th>
<th>coordinates.latitude</th>
<th>coordinates.longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>BELAL01</td>
<td>51.23619</td>
<td>4.38522</td>
</tr>
<tr>
<td>BELHB23</td>
<td>51.17030</td>
<td>4.34100</td>
</tr>
<tr>
<td>BELLD01</td>
<td>51.10998</td>
<td>5.00486</td>
</tr>
<tr>
<td>BELLD02</td>
<td>51.12038</td>
<td>5.02155</td>
</tr>
<tr>
<td>BELR833</td>
<td>51.32766</td>
<td>4.36226</td>
</tr>
</tbody>
</table>
```

1.4. Tutorials  49
Note: The stations used in this example (FR04014, BETR801 and London Westminster) are just three entries enlisted in the metadata table. We only want to add the coordinates of these three to the measurements table, each on the corresponding rows of the \textit{air\_quality} table.

\begin{verbatim}
In [19]: air_quality.head()
Out[19]:
   date.utc     location parameter  value
0 2019-05-07 01:00:00+00:00 London Westminster  no2 23.0
1 2019-05-07 01:00:00+00:00   FR04014  no2 25.0
100 2019-05-07 01:00:00+00:00     BETR801  pm25 12.5
1098 2019-05-07 01:00:00+00:00  BETR801  no2 50.5
1109 2019-05-07 01:00:00+00:00 London Westminster  pm25  8.0
\end{verbatim}

\begin{verbatim}
In [20]: air_quality = pd.merge(air_quality, stations_coord, how="left", on="location")
In [21]: air_quality.head()
Out[21]:
   date.utc     location parameter  value  coordinates.latitude  coordinates.longitude
0 2019-05-07 01:00:00+00:00 London Westminster  no2 23.0  51.49467        -0.13193
1 2019-05-07 01:00:00+00:00   FR04014  no2 25.0  48.83724         2.39390
2 2019-05-07 01:00:00+00:00   FR04014  no2 25.0  48.83722         2.39390
3 2019-05-07 01:00:00+00:00     BETR801  pm25 12.5  51.20966         4.43182
4 2019-05-07 01:00:00+00:00  BETR801  no2 50.5  51.20966         4.43182
\end{verbatim}

Using the \texttt{merge()} function, for each of the rows in the \texttt{air\_quality} table, the corresponding coordinates are added from the \texttt{air\_quality\_stations\_coord} table. Both tables have the column \texttt{location} in common which is used as a key to combine the information. By choosing the \texttt{left} join, only the locations available in the \texttt{air\_quality} (left) table, i.e. FR04014, BETR801 and London Westminster, end up in the resulting table. The \texttt{merge} function supports multiple join options similar to database-style operations.

Add the parameter full description and name, provided by the parameters metadata table, to the measurements table.

\begin{verbatim}
Warning: The air quality parameters metadata are stored in a data file \texttt{air\_quality\_parameters.csv}, downloaded using the \texttt{py-openaq} package.
\end{verbatim}

\begin{verbatim}
In [22]: air_quality_parameters = pd.read_csv("data/air_quality_parameters.csv")
In [23]: air_quality_parameters.head()
Out[23]:
   id    description    name
0  bc  Black Carbon  BC
1  co  Carbon Monoxide  CO
2  no2 Nitrogen Dioxide  NO2
3   o3     Ozone     O3
\end{verbatim}
Compared to the previous example, there is no common column name. However, the `parameter` column in the `air_quality` table and the `id` column in the `air_quality_parameters` both provide the measured variable in a common format. The `left_on` and `right_on` arguments are used here (instead of just `on`) to make the link between the two tables.

pandas supports also inner, outer, and right joins. More information on join/merge of tables is provided in the user guide section on database style merging of tables. Or have a look at the comparison with SQL page.

- Multiple tables can be concatenated both column-wise and row-wise using the `concat` function.

- For database-like merging/joining of tables, use the `merge` function.

See the user guide for a full description of the various facilities to combine data tables.

```
In [1]: import pandas as pd
In [2]: import matplotlib.pyplot as plt

For this tutorial, air quality data about $\text{NO}_2$ and Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package. The `air_quality_no2_long.csv` data set provides $\text{NO}_2$ values for the measurement stations `FR04014`, `BETR801` and `London Westminster` in respectively Paris, Antwerp and London.

```
In [3]: air_quality = pd.read_csv("data/air_quality_no2_long.csv")
In [4]: air_quality = air_quality.rename(columns={"date.utc": "datetime"})
In [5]: air_quality.head()
Out[5]:
   city   country         datetime location parameter  value  unit
0  Paris     FR  2019-06-21 00:00:00+00:00   FR04014  no2  20.0  µg/m³
```
How to handle time series data with ease?

Using pandas datetime properties

I want to work with the dates in the column `datetime` as datetime objects instead of plain text

```python
In [7]: air_quality["datetime"] = pd.to_datetime(air_quality["datetime"])  
```

```plaintext
Out[8]:  
```

Initially, the values in `datetime` are character strings and do not provide any datetime operations (e.g. extract the year, day of the week,...). By applying the `to_datetime` function, pandas interprets the strings and converts these to datetime (i.e. `datetime64[ns, UTC]`) objects. In pandas we call these datetime objects similar to `datetime.datetime` from the standard library as `pandas.Timestamp`.

Note: As many data sets do contain datetime information in one of the columns, pandas input function like `pandas.read_csv()` and `pandas.read_json()` can do the transformation to dates when reading the data using the `parse_dates` parameter with a list of the columns to read as Timestamp:

```python
pd.read_csv("./data/air_quality_no2_long.csv", parse_dates=["datetime"])  
```

Why are these `pandas.Timestamp` objects useful? Let’s illustrate the added value with some example cases.

What is the start and end date of the time series data set we are working with?

```python
In [9]: air_quality["datetime"].min(), air_quality["datetime"].max()  
```

```plaintext
Out[9]:  
```

52 Chapter 1. Getting started
Using `pandas.Timestamp` for datetimes enables us to calculate with date information and make them comparable. Hence, we can use this to get the length of our time series:

```python
In [10]: air_quality["datetime"].max() - air_quality["datetime"].min()
Out[10]: Timedelta('44 days 23:00:00')
```

The result is a `pandas.Timedelta` object, similar to `datetime.timedelta` from the standard Python library and defining a time duration.

The various time concepts supported by pandas are explained in the user guide section on `time related concepts`.

I want to add a new column to the DataFrame containing only the month of the measurement:

```python
In [11]: air_quality["month"] = air_quality["datetime"].dt.month

In [12]: air_quality.head()
Out[12]:
<table>
<thead>
<tr>
<th></th>
<th>city</th>
<th>country</th>
<th>datetime</th>
<th>location</th>
<th>parameter</th>
<th>value</th>
<th>unit</th>
<th>month</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Paris</td>
<td>FR</td>
<td>2019-06-21</td>
<td>FR04014</td>
<td>no2</td>
<td>20.0</td>
<td>µg/m³</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>Paris</td>
<td>FR</td>
<td>2019-06-20</td>
<td>FR04014</td>
<td>no2</td>
<td>21.8</td>
<td>µg/m³</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Paris</td>
<td>FR</td>
<td>2019-06-20</td>
<td>FR04014</td>
<td>no2</td>
<td>26.5</td>
<td>µg/m³</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Paris</td>
<td>FR</td>
<td>2019-06-20</td>
<td>FR04014</td>
<td>no2</td>
<td>24.9</td>
<td>µg/m³</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Paris</td>
<td>FR</td>
<td>2019-06-20</td>
<td>FR04014</td>
<td>no2</td>
<td>21.4</td>
<td>µg/m³</td>
<td>6</td>
</tr>
</tbody>
</table>
```

By using `Timestamp` objects for dates, a lot of time-related properties are provided by pandas. For example the `month`, but also `year`, `weekofyear`, `quarter`,... All of these properties are accessible by the `dt` accessor.

An overview of the existing date properties is given in the `time and date components overview table`. More details about the `dt` accessor to return datetime like properties are explained in a dedicated section on the `dt accessor`.

What is the average \( \text{NO}_2 \) concentration for each day of the week for each of the measurement locations?

```python
In [13]: air_quality.groupby(
    ....:     [air_quality["datetime"].dt.weekday, "location"])["value"].mean()
    ....:
Out[13]:
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>datetime</td>
<td>location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>BETR801</td>
<td>27.875000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FR04014</td>
<td>24.856250</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>London Westminster</td>
<td>23.969697</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BETR801</td>
<td>22.214286</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FR04014</td>
<td>30.993359</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>London Westminster</td>
<td>24.214286</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FR04014</td>
<td>25.266154</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>London Westminster</td>
<td>24.977612</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BETR801</td>
<td>21.896552</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FR04014</td>
<td>23.274306</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>London Westminster</td>
<td>24.859155</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Name: value, Length: 21, dtype: float64
```

Remember the split-apply-combine pattern provided by `groupby` from the `tutorial on statistics calculation`? Here, we want to calculate a given statistic (e.g. mean \( \text{NO}_2 \)) for each weekday and for each measurement location. To group on weekdays, we use the datetime property `weekday` (with Monday=0 and Sunday=6) of pandas `Timestamp`, which is also accessible by the `dt` accessor. The grouping on both locations and weekdays can be done to split the calculation of the mean on each of these combinations.
Plot the typical $NO_2$ pattern during the day of our time series of all stations together. In other words, what is the average value for each hour of the day?

```
In [14]: fig, axs = plt.subplots(figsize=(12, 4))
In [15]: air_quality.groupby(air_quality['datetime'].dt.hour)['value'].mean().plot(
      .....:     kind='bar', rot=0, ax=axs
      .....: )
Out[15]: <AxesSubplot:xlabel='datetime'>
In [16]: plt.xlabel("Hour of the day"); # custom x label using matplotlib
In [17]: plt.ylabel("$NO_2 (\mu g/m^3)$");
```

![Graph showing $NO_2$ pattern during the day.]

Similar to the previous case, we want to calculate a given statistic (e.g. mean $NO_2$) for each hour of the day and we can use the split-apply-combine approach again. For this case, we use the datetime property hour of pandas Timestamp, which is also accessible by the dt accessor.

**Datetime as index**

In the *tutorial on reshaping, pivot()* was introduced to reshape the data table with each of the measurements locations as a separate column:

```
In [18]: no_2 = air_quality.pivot(index="datetime", columns="location", values="value")
In [19]: no_2.head()
Out[19]:

<table>
<thead>
<tr>
<th>location</th>
<th>BETR801</th>
<th>FR04014</th>
<th>London Westminster</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019-05-07 01:00:00+00:00</td>
<td>50.5</td>
<td>25.0</td>
<td>23.0</td>
</tr>
<tr>
<td>2019-05-07 02:00:00+00:00</td>
<td>45.0</td>
<td>27.7</td>
<td>19.0</td>
</tr>
<tr>
<td>2019-05-07 03:00:00+00:00</td>
<td>NaN</td>
<td>50.4</td>
<td>19.0</td>
</tr>
<tr>
<td>2019-05-07 04:00:00+00:00</td>
<td>NaN</td>
<td>61.9</td>
<td>16.0</td>
</tr>
<tr>
<td>2019-05-07 05:00:00+00:00</td>
<td>NaN</td>
<td>72.4</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

---

**Danger:** As we are working with a very short time series in these examples, the analysis does not provide a long-term representative result!
Note: By pivoting the data, the datetime information became the index of the table. In general, setting a column as an index can be achieved by the `set_index` function.

Working with a datetime index (i.e. `DatetimeIndex`) provides powerful functionalities. For example, we do not need the `dt` accessor to get the time series properties, but have these properties available on the index directly:

```python
In [20]: no_2.index.year, no_2.index.weekday
Out[20]:
(Int64Index([2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019,
            ...
            dtype='int64', name='datetime', length=1033),
Int64Index([1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            ...
            3, 3, 3, 3, 3, 3, 3, 3, 3, 4],
            dtype='int64', name='datetime', length=1033))
```

Some other advantages are the convenient subsetting of time period or the adapted time scale on plots. Let's apply this on our data.

Create a plot of the $NO_2$ values in the different stations from the 20th of May till the end of 21st of May

```python
In [21]: no_2["2019-05-20":"2019-05-21"].plot();
```
By providing a **string that parses to a datetime**, a specific subset of the data can be selected on a `DatetimeIndex`. More information on the `DatetimeIndex` and the slicing by using strings is provided in the section on *time series indexing*.

### Resample a time series to another frequency

Aggregate the current hourly time series values to the monthly maximum value in each of the stations.

```
In [22]: monthly_max = no_2.resample("M").max()
In [23]: monthly_max
Out[23]:
<table>
<thead>
<tr>
<th>location</th>
<th>BETR801</th>
<th>FR04014</th>
<th>London Westminster</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019-05-31</td>
<td>74.5</td>
<td>97.0</td>
<td>97.0</td>
</tr>
<tr>
<td>2019-06-30</td>
<td>52.5</td>
<td>84.7</td>
<td>52.0</td>
</tr>
</tbody>
</table>
```

A very powerful method on time series data with a datetime index, is the ability to `resample()` time series to another frequency (e.g., converting secondly data into 5-minutely data).

The `resample()` method is similar to a groupby operation:

- it provides a time-based grouping, by using a string (e.g. M, 5H,...) that defines the target frequency
- it requires an aggregation function such as `mean`, `max`, ...

An overview of the aliases used to define time series frequencies is given in the *offset aliases overview table*. When defined, the frequency of the time series is provided by the `freq` attribute:

```
In [24]: monthly_max.index.freq
Out[24]: MonthEnd
```

Make a plot of the daily mean \( NO_2 \) value in each of the stations.

```
In [25]: no_2.resample("D").mean().plot(style="-o", figsize=(10, 5));
```
More details on the power of time series resampling is provided in the user guide section on `resampling`.

- Valid date strings can be converted to datetime objects using `to_datetime` function or as part of read functions.
- Datetime objects in pandas support calculations, logical operations and convenient date-related properties using the `dt` accessor.
- A `DatetimeIndex` contains these date-related properties and supports convenient slicing.
- `Resample` is a powerful method to change the frequency of a time series.

A full overview on time series is given on the pages on `time series and date functionality`.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.
How to manipulate textual data?

Make all name characters lowercase.

```
In [4]: titanic["Name"].str.lower()
```

```
Out[4]:
0                   braund, mr. owen harris
1  cumings, mrs. john bradley (florence briggs th...
2                    heikkinen, miss. laina
3  futrelle, mrs. jacques heath (lily may peel)
4  allan, mr. william henry
    ...                    montvila, rev. juozas
886                  graham, miss. margaret edith
887                        johnston, miss. catherine helen "carrie"
889                        behr, mr. karl howell
890                   dooley, mr. patrick
Name: Name, Length: 891, dtype: object
```

To make each of the strings in the Name column lowercase, select the Name column (see the tutorial on selection of data), add the str accessor and apply the lower method. As such, each of the strings is converted element-wise.

Similar to datetime objects in the time series tutorial having a dt accessor, a number of specialized string methods are available when using the str accessor. These methods have in general matching names with the equivalent built-in string methods for single elements, but are applied element-wise (remember element-wise calculations?) on each of the values of the columns.

Create a new column Surname that contains the surname of the passengers by extracting the part before the comma.

```
In [5]: titanic["Name"].str.split(",")
```

```
Out[5]:
0              [Braund, Mr. Owen Harris]
1  [Cumings, Mrs. John Bradley (Florence Briggs ...]
2  [Heikkinen, Miss. Laina]
```
Using the `Series.str.split()` method, each of the values is returned as a list of 2 elements. The first element is the part before the comma and the second element is the part after the comma.

```python
In [6]: titanic['Surname'] = titanic['Name'].str.split(',').str.get(0)
In [7]: titanic['Surname']
Out[7]:
0     Braund
1    Cumings
2   Heikkinen
3    Futrelle
4      Allen
     ...  
886  Montvila
887    Graham
888    Johnston
889      Behr
890    Dooley
Name: Surname, Length: 891, dtype: object
```

As we are only interested in the first part representing the surname (element 0), we can again use the `str` accessor and apply `Series.str.get()` to extract the relevant part. Indeed, these string functions can be concatenated to combine multiple functions at once!

More information on extracting parts of strings is available in the user guide section on *splitting and replacing strings*.

Extract the passenger data about the countesses on board of the Titanic.

```python
In [8]: titanic['Name'].str.contains("Countess")
Out[8]:
0   False
1   False
2   False
3   False
4   False
   ...  
886  False
887  False
888  False
889  False
890  False
Name: Name, Length: 891, dtype: bool
```
The string method `Series.str.contains()` checks for each of the values in the column `Name` if the string contains the word `Countess` and returns for each of the values `True` (Countess is part of the name) or `False` (Countess is not part of the name). This output can be used to subselect the data using conditional (boolean) indexing introduced in the subsetting of data tutorial. As there was only one countess on the Titanic, we get one row as a result.

Note: More powerful extractions on strings are supported, as the `Series.str.contains()` and `Series.str.extract()` methods accept regular expressions, but out of scope of this tutorial.

More information on extracting parts of strings is available in the user guide section on string matching and extracting.

Which passenger of the Titanic has the longest name?

To get the longest name we first have to get the lengths of each of the names in the `Name` column. By using pandas string methods, the `Series.str.len()` function is applied to each of the names individually (element-wise).

Next, we need to get the corresponding location, preferably the index label, in the table for which the name length is the largest. The `idxmax()` method does exactly that. It is not a string method and is applied to integers, so no `str` is used.

Based on the index name of the row (307) and the column (Name), we can do a selection using the `loc` operator, introduced in the tutorial on subsetting.
In the “Sex” column, replace values of “male” by “M” and values of “female” by “F”.

```python
In [13]: titanic["Sex_short"] = titanic["Sex"].replace({"male": "M", "female": "F"})

In [14]: titanic["Sex_short"]
Out[14]:
0    M
1    F
2    F
3    F
4    M
   ..
886   M
887   F
888   F
889   M
890   M
Name: Sex_short, Length: 891, dtype: object
```

Whereas `replace()` is not a string method, it provides a convenient way to use mappings or vocabularies to translate certain values. It requires a dictionary to define the mapping `{from : to}`.

**Warning:** There is also a `replace()` method available to replace a specific set of characters. However, when having a mapping of multiple values, this would become:

```python
titanic["Sex_short"] = titanic["Sex"].str.replace("female", "F")
titanic["Sex_short"] = titanic["Sex_short"].str.replace("male", "M")
```

This would become cumbersome and easily lead to mistakes. Just think (or try out yourself) what would happen if those two statements are applied in the opposite order...

- String methods are available using the `str` accessor.
- String methods work element-wise and can be used for conditional indexing.
- The `replace` method is a convenient method to convert values according to a given dictionary.

A full overview is provided in the user guide pages on *working with text data*.

### 1.4.4 Comparison with other tools

#### Comparison with R / R libraries

Since pandas aims to provide a lot of the data manipulation and analysis functionality that people use R for, this page was started to provide a more detailed look at the R language and its many third party libraries as they relate to pandas. In comparisons with R and CRAN libraries, we care about the following things:

- **Functionality / flexibility:** what can/cannot be done with each tool
- **Performance:** how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use:** Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these R packages.
For transfer of DataFrame objects from pandas to R, one option is to use HDF5 files, see External compatibility for an example.

**Quick reference**

We’ll start off with a quick reference guide pairing some common R operations using dplyr with pandas equivalents.

**Querying, filtering, sampling**

<table>
<thead>
<tr>
<th>R</th>
<th>pandas</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim(df)</td>
<td>df.shape</td>
</tr>
<tr>
<td>head(df)</td>
<td>df.head()</td>
</tr>
<tr>
<td>slice(df, 1:10)</td>
<td>df.iloc[:9]</td>
</tr>
<tr>
<td>filter(df, col1 == 1, col2 == 1)</td>
<td>df.query('col1 == 1 &amp; col2 == 1')</td>
</tr>
<tr>
<td>df[df$col1 == 1 &amp; df$col2 == 1]</td>
<td>df[(df.col1 == 1) &amp; (df.col2 == 1)]</td>
</tr>
<tr>
<td>select(df, col1, col2)</td>
<td>df[['col1', 'col2']]</td>
</tr>
<tr>
<td>select(df, col1:col3)</td>
<td>df.loc[:, 'col1':'col3']</td>
</tr>
<tr>
<td>select(df, -(col1:col3))</td>
<td>df.drop(cols_to_drop, axis=1) but see'</td>
</tr>
<tr>
<td>distinct(select(df, col1))</td>
<td>df[['col1']].drop_duplicates()</td>
</tr>
<tr>
<td>distinct(select(df, col1, col2))</td>
<td>df[['col1', 'col2']].drop_duplicates()</td>
</tr>
<tr>
<td>sample_n(df, 10)</td>
<td>df.sample(n=10)</td>
</tr>
<tr>
<td>sample_frac(df, 0.01)</td>
<td>df.sample(frac=0.01)</td>
</tr>
</tbody>
</table>

**Sorting**

<table>
<thead>
<tr>
<th>R</th>
<th>pandas</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrange(df, col1, col2)</td>
<td>df.sort_values(['col1', 'col2'])</td>
</tr>
<tr>
<td>arrange(df, desc(col1))</td>
<td>df.sort_values('col1', ascending=False)</td>
</tr>
</tbody>
</table>

**Transforming**

<table>
<thead>
<tr>
<th>R</th>
<th>pandas</th>
</tr>
</thead>
<tbody>
<tr>
<td>select(df, col_one = col1)</td>
<td>df.rename(columns={'col1': 'col_one'})['col_one']</td>
</tr>
<tr>
<td>rename(df, col_one = col1)</td>
<td>df.rename(columns={'col1': 'col_one'})</td>
</tr>
<tr>
<td>mutate(df, c=a-b)</td>
<td>df.assign(c=df['a']-df['b'])</td>
</tr>
</tbody>
</table>

1. R’s shorthand for a subrange of columns (select(df, col1:col3)) can be approached cleanly in pandas, if you have the list of columns, for example df[cols[1:3]] or df.drop(cols[1:3]), but doing this by column name is a bit messy.
Grouping and summarizing

<table>
<thead>
<tr>
<th>R</th>
<th>pandas</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>summary(df)</code></td>
<td><code>df.describe()</code></td>
</tr>
<tr>
<td><code>gdf &lt;- group_by(df, col1)</code></td>
<td><code>gdf = df.groupby('col1')</code></td>
</tr>
<tr>
<td><code>summarise(gdf, avg=mean(col1, na.rm=TRUE))</code></td>
<td><code>gdf.groupby('col1').agg({'col1': 'mean'})</code></td>
</tr>
<tr>
<td><code>summarise(gdf, total=sum(col1))</code></td>
<td><code>gdf.groupby('col1').sum()</code></td>
</tr>
</tbody>
</table>

**Base R**

**Slicing with R's c**

R makes it easy to access `data.frame` columns by name

```r
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```r
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in pandas is straightforward

```python
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list("abc"))
In [2]: df["a", "c"]
```

```
Out[2]:
          a         c
0  0.469112 -1.50906
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892
```

```python
In [3]: df.loc[:, ["a", "c"]]
```

```
Out[3]:
          a         c
0  0.469112 -1.50906
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892
```

(continues on next page)
Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list("abcdefg")

In [5]: n = 30

In [6]: columns = named + np.arange(len(named), n).tolist()

In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)

In [8]: df.iloc[:, np.r_[0:10, 24:30]]
```

```
Out[8]:
      a     b     c     d     e     f   ...    24    25
     ---   ---   ---   ---   ---   ---   ...   ---   ---
     0 -1.34431  0.84488  1.07577 -0.10905 1.64356 ... -1.17029 -0.22617
     ... -0.41083  0.81385  0.13200 -0.82731  0.61976 ... 0.41083  0.81385
     1 -0.07647 -1.18768  1.13012 -1.43673  1.60792 ...  0.95972 -1.10336 ...
     ...  0.14974  0.73233  0.68773 0.14974  0.73233 ...  0.68773 0.14974
     2  0.17644  0.40331 -0.15495  0.30162 -2.17986 ... -1.36984  0.08484 ...
     ... -0.49366  0.60018  0.27423 -0.49366  0.60018 ...  0.27423 -0.49366
     3  0.13288 -0.02368  2.41018  1.45052  0.20605 ... -2.48478 -0.28146 ...
     ...  0.10912  1.12620 -0.97734  0.10912  1.12620 ... -0.97734  0.10912
     4  1.47407 -0.06403 -1.28278  0.78184 -1.07136 ... -1.19707 -1.06696 ...
     ... -0.30342 -0.85845  0.30696 -0.30342 -0.85845 ...  0.30696 -0.30342
     ...  0.72524  0.62461 -0.14119 -0.14119  0.72524 ... -0.14119  0.72524 ...
     25  1.49213 -0.06819  0.68146  1.22183 -0.43435  1.20481 ...  1.94451  0.04234 ...
     ...  0.42857  0.88069  0.48764  0.42857  0.88069 ...  0.48764  0.42857
     26  0.72523  0.62460 -0.14118 -0.14394 -0.32816  2.09508 ... -0.84618  1.19062 ...
     ...  1.42401 -0.71711 -0.71711  1.42401 ...  0.71711  1.42401 ...
     27  1.26242  1.95065  0.30103 -0.93385  0.81495  0.18143 ... -1.34181  0.33428 ...
     ...  0.16222  0.28260  0.14583  0.16222  0.28260 ...  0.14583  0.16222 ...
     28  1.58574 -0.89973  0.92194 -0.21176 -0.05918  0.05830 ...  0.40362  0.02660 ...
     ... -0.24081  0.57722 -1.08841  0.32668  0.57722 ... -1.08841  0.32668 ...
     29 -0.98625  0.16973 -1.15809  1.01967  0.64604  0.91739 ... -1.20924 -0.67146 ...
     ...  0.33287 -2.01308 -1.60254  0.33287 -2.01308 ... -1.60254  0.33287 [...30 rows x 16 columns]
```
**aggregate**

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called `df` and splitting it into groups `by1` and `by2`:

```r
df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)
```

The `groupby()` method is similar to base R `aggregate` function.

```python
In [9]: df = pd.DataFrame(
    ...: {  
    ...:     "v1": [1, 3, 5, 7, 8, 3, 5, np.nan, 4, 5, 7, 9],  
    ...:     "v2": [11, 33, 55, 77, 88, 33, 55, np.nan, 44, 55, 77, 99],  
    ...:     "by1": ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],  
    ...:     "by2": [
    ...:         "wet",  
    ...:         "dry",  
    ...:         99,  
    ...:         95,  
    ...:         np.nan,  
    ...:         "damp",  
    ...:         95,  
    ...:         99,  
    ...:         "red",  
    ...:         99,  
    ...:         np.nan,  
    ...:         np.nan,  
    ...:     ],  
    ...: },  
    ...: )

In [10]: g = df.groupby(["by1", "by2"])  
In [11]: g["v1", "v2"]).mean()
Out[11]:

<table>
<thead>
<tr>
<th>by1</th>
<th>by2</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99</td>
<td>5.0</td>
<td>55.0</td>
</tr>
<tr>
<td>1</td>
<td>95</td>
<td>5.0</td>
<td>55.0</td>
</tr>
<tr>
<td>2</td>
<td>95</td>
<td>7.0</td>
<td>77.0</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>big</td>
<td>damp</td>
<td>3.0</td>
<td>33.0</td>
</tr>
<tr>
<td>blue</td>
<td>dry</td>
<td>3.0</td>
<td>33.0</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>4.0</td>
<td>44.0</td>
</tr>
<tr>
<td>wet</td>
<td></td>
<td>1.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>
```

For more details and examples see the groupby documentation.
match / %in%

A common way to select data in R is using %in% which is defined using the function `match`. The operator %in% is used to return a logical vector indicating if there is a match or not:

```r
s <- 0:4
s %in% c(2,4)
```

The `isin()` method is similar to R %in% operator:

```python
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)
In [13]: s.isin([2, 4])
Out[13]:
0   False
1   False
2   True
3   False
4   True
dtype: bool
```

The `match` function returns a vector of the positions of matches of its first argument in its second:

```r
s <- 0:4
match(s, c(2,4))
```

For more details and examples see the reshaping documentation.

tapply

tapply is similar to aggregate, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called `baseball`, and retrieving information based on the array `team`:

```r
baseball <-
data.frame(team = gl(5, 5,
    labels = paste("Team", LETTERS[1:5])),
    player = sample(letters, 25),
    batting.average = runif(25, .200, .400))
tapply(baseball$batting.average, baseball.example$team,
    max)
```

In pandas we may use `pivot_table()` method to handle this:

```python
In [14]: import random
In [15]: import string
In [16]: baseball = pd.DataFrame(
 ....:     {
 ....:         "team": ["team %d" % (x + 1) for x in range(5)] * 5,
 ....:         "player": random.sample(list(string.ascii_lowercase), 25),
 ....:         "batting avg": np.random.uniform(0.200, 0.400, 25),
 ....:     }
```

(continues on next page)
In [17]: baseball.pivot_table(values="batting avg", columns="team", aggfunc=np.max)
Out[17]:
           team 1      team 2      team 3      team 4      team 5
batting avg  0.352134  0.295327  0.397191  0.394457  0.396194

For more details and examples see *the reshaping documentation*.

**subset**

The *query()* method is similar to the base R *subset* function. In R you might want to get the rows of a *data.frame* where one column's values are less than another column's values:

```python
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,] # note the comma
```

In pandas, there are a few ways to perform subsetting. You can use *query()* or pass an expression as if it were an index/slice as well as standard boolean indexing:

```python
In [18]: df = pd.DataFrame({"a": np.random.randn(10), "b": np.random.randn(10)})
In [19]: df.query("a <= b")
Out[19]:
a   b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550
In [20]: df[df["a"] <= df["b"]]
Out[20]:
a   b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550
In [21]: df.loc[df["a"] <= df["b"]]
Out[21]:
a   b
1  0.174950  0.552887
```

(continues on next page)
An expression using a data.frame called df in R with the columns a and b would be evaluated using with like so:

```r
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b  # same as the previous expression
```

In pandas the equivalent expression, using the `eval()` method, would be:

```python
In [22]: df = pd.DataFrame({"a": np.random.randn(10), "b": np.random.randn(10)})
In [23]: df.eval("a + b")
Out[23]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6    0.799331
7   -2.377245
8    2.104677
dtype: float64
```

```python
In [24]: df["a"] + df["b"]  # same as the previous expression
Out[24]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6    0.799331
7   -2.377245
8    2.104677
dtype: float64
```

In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see the `eval` documentation.
plyr

plyr is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, a for arrays, l for lists, and d for data.frame. The table below shows how these data structures could be mapped in Python.

<table>
<thead>
<tr>
<th>R</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>lists</td>
<td>dictionary or list of objects</td>
</tr>
<tr>
<td>data.frame</td>
<td>dataframe</td>
</tr>
</tbody>
</table>

ddply

An expression using a data.frame called df in R where you want to summarize x by month:

```r
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
  mean = round(mean(x), 2),
  sd = round(sd(x), 2))
```

In pandas the equivalent expression, using the `groupby()` method, would be:

```python
In [25]: df = pd.DataFrame(
    ...:    {
    ...:        "x": np.random.uniform(1.0, 168.0, 120),
    ...:        "y": np.random.uniform(7.0, 334.0, 120),
    ...:        "z": np.random.uniform(1.7, 20.7, 120),
    ...:        "month": [5, 6, 7, 8] * 30,
    ...:        "week": np.random.randint(1, 4, 120),
    ...:    }
    ...: )
    ...

In [26]: grouped = df.groupby(['month', 'week'])

In [27]: grouped["x"].agg([np.mean, np.std])
Out[27]:

<table>
<thead>
<tr>
<th>mean</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>week</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
```

(continues on next page)
For more details and examples see the groupby documentation.

**reshape / reshape2**

**meltarray**

An expression using a 3 dimensional array called a in R where you want to melt it into a data.frame:

```r
a <- array(c(1:23, NA, c(2, 3, 4)), c(2, 3, 4))
data.frame(melt(a))
```

In Python, since `a` is a list, you can simply use list comprehension.

```python
In [28]: a = np.array(list(range(1, 24)) + [np.NAN]).reshape(2, 3, 4)
In [29]: pd.DataFrame([tuple(list(x) + [val]) for x, val in np.ndenumerate(a)])
Out[29]:
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>20.0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>21.0</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>22.0</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>23.0</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

[24 rows x 4 columns]

**meltlist**

An expression using a list called `a` in R where you want to melt it into a data.frame:

```r
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.
In [30]: a = list(enumerate(list(range(1, 5)) + [np.NAN]))

In [31]: pd.DataFrame(a)
Out[31]:
   0   1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  NaN

For more details and examples see the Into to Data Structures documentation.

melt
df

An expression using a data.frame called cheese in R where you want to reshape the data.frame:

cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
)
melt(cheese, id=c("first", "last"))

In Python, the melt() method is the R equivalent:

In [32]: cheese = pd.DataFrame(
       ....: { 
       ....:     "first": ["John", "Mary"],
       ....:     "last": ["Doe", "Bo"],
       ....:     "height": [5.5, 6.0],
       ....:     "weight": [130, 150],
       ....:   }
       ....: )
       ....:

In [33]: pd.melt(cheese, id_vars=["first", "last" ])
Out[33]:
   first last variable value
0  John Doe  height  5.5
1  Mary Bo  height  6.0
2  John Doe  weight 130.0
3  Mary Bo  weight 150.0

In [34]: cheese.set_index(["first", "last"]).stack()  # alternative way
Out[34]:
   first last
John Doe  height  5.5
          weight 130.0
Mary Bo  height  6.0
          weight 150.0

(continues on next page)
For more details and examples see the reshaping documentation.

**cast**

In R, `acast` is an expression using a `data.frame` called `df` in R to cast into a higher dimensional array:

```r
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7),4),
  week = rep(c(1,2),6)
)
mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python, the best way is to make use of `pivot_table()`:

```python
In [35]: df = pd.DataFrame(
    ....:     {
    ....:         "x": np.random.uniform(1.0, 168.0, 12),
    ....:         "y": np.random.uniform(7.0, 334.0, 12),
    ....:         "z": np.random.uniform(1.7, 20.7, 12),
    ....:         "month": [5, 6, 7] * 4,
    ....:         "week": [1, 2] * 6,
    ....:     }
    ....: )

In [36]: mdf = pd.melt(df, id_vars=["month", "week"])

In [37]: pd.pivot_table(
    ....:     mdf,
    ....:     values="value",
    ....:     index=["variable", "week"],
    ....:     columns=["month"],
    ....:     aggfunc=np.mean,
    ....: )

Out[37]:

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>93.888747</td>
<td>98.762034</td>
<td>55.219673</td>
</tr>
<tr>
<td></td>
<td>94.391427</td>
<td>38.112932</td>
<td>83.942781</td>
</tr>
<tr>
<td>y</td>
<td>94.306912</td>
<td>279.454811</td>
<td>227.840449</td>
</tr>
<tr>
<td></td>
<td>87.392662</td>
<td>193.028166</td>
<td>173.899260</td>
</tr>
<tr>
<td>z</td>
<td>11.016009</td>
<td>10.079307</td>
<td>16.170549</td>
</tr>
<tr>
<td></td>
<td>8.476111</td>
<td>17.638509</td>
<td>19.003494</td>
</tr>
</tbody>
</table>
```
Similarly for `dcast` which uses a `data.frame` called `df` in R to aggregate information based on `Animal` and `FeedType`:

```r
df <- data.frame(
    Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
               'Animal2', 'Animal3'),
    FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
    Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))
```

Python can approach this in two different ways. Firstly, similar to above using `pivot_table()`:

```python
In [38]: df = pd.DataFrame(
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       
       In [39]: df.pivot_table(values="Amount", index="Animal", columns="FeedType", aggfunc="sum")
Out[39]:
FeedType  A  B
Animal   
Animal1  10.0 5.0
Animal2  2.0 13.0
Animal3  6.0 NaN
```

The second approach is to use the `groupby()` method:

```python
In [40]: df.groupby(["Animal", "FeedType"])["Amount"].sum()
Out[40]:
Animal  FeedType
Animal1 A 10
       B 5
Animal2 A 2
       B 13
Animal3 A 6
Name: Amount, dtype: int64
```

For more details and examples see [the reshaping documentation](#) or [the groupby documentation](#).
factor

pandas has a data type for categorical data.

\[
\text{cut}(c(1,2,3,4,5,6), 3) \\
\text{factor}(c(1,2,3,2,2,3))
\]

In pandas this is accomplished with `pd.cut` and `astype("category")`:

\[
\text{In [41]: } \text{pd.cut(pd.Series([1, 2, 3, 4, 5, 6]), 3)} \\
\text{Out[41]:} \\
0 (0.995, 2.667] \\
1 (0.995, 2.667] \\
2 (2.667, 4.333] \\
3 (2.667, 4.333] \\
4 (4.333, 6.0] \\
5 (4.333, 6.0] \\
dtype: category \\
Categories (3, interval[float64, right]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.] \\
\text{In [42]: } \text{pd.Series([1, 2, 3, 2, 2, 3]).astype("category")} \\
\text{Out[42]:} \\
0 1 \\
1 2 \\
2 3 \\
3 2 \\
4 2 \\
5 3 \\
dtype: category \\
Categories (3, int64): [1, 2, 3]
\]

For more details and examples see categorical introduction and the API documentation. There is also a documentation regarding the differences to R’s factor.

Comparison with SQL

Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you’re new to pandas, you might want to first read through 10 Minutes to pandas to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

\[
\text{In [1]: } \text{import pandas as pd} \\
\text{In [2]: } \text{import numpy as np}
\]

Most of the examples will utilize the tips dataset found within pandas tests. We’ll read the data into a DataFrame called `tips` and assume we have a database table of the same name and structure.
...:  "/pandas/main/pandas/tests/io/data/csv/tips.csv"
...:  )

In [4]: tips = pd.read_csv(url)
In [5]: tips
Out[5]:

<table>
<thead>
<tr>
<th></th>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>21.01</td>
<td>3.50</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>23.68</td>
<td>3.31</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>24.59</td>
<td>3.61</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>239</td>
<td>29.03</td>
<td>5.92</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>240</td>
<td>27.18</td>
<td>2.00</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>241</td>
<td>22.67</td>
<td>2.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>242</td>
<td>17.82</td>
<td>1.75</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>243</td>
<td>18.78</td>
<td>3.00</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Dinner</td>
<td>2</td>
</tr>
</tbody>
</table>

[244 rows x 7 columns]

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes “stick”, you’ll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

**Note:** You will see an inplace=True keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```

Its use is discouraged. More information.
SELECT

In SQL, selection is done using a comma-separated list of columns you'd like to select (or a * to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips["total_bill", "tip", "smoker", "time"]
```

```
Out[6]:
    total_bill  tip  smoker  time
0     16.99  1.01    No  Dinner
1     10.34  1.66    No  Dinner
2     21.01  3.50    No  Dinner
3     23.68  3.31    No  Dinner
4     24.59  3.61    No  Dinner
..     ...    ...     ...       ...
239   29.03  5.92    No  Dinner
240   27.18  2.00    Yes Dinner
241   22.67  2.00    Yes Dinner
242   17.82  1.75    No  Dinner
243   18.78  3.00    No  Dinner

[244 rows x 4 columns]
```

Calling the DataFrame without the list of column names would display all columns (akin to SQL's *).

In SQL, you can add a calculated column:

```
SELECT *, tip/total_bill as tip_rate
FROM tips;
```

With pandas, you can use the `DataFrame.assign()` method of a DataFrame to append a new column:

```
In [7]: tips.assign(tip_rate=tips["tip"] / tips["total_bill")
```

```
Out[7]:
    total_bill  tip  sex  smoker  day  time  size  tip_rate
0       16.99  1.01 Female    No  Sun  Dinner  2  0.059447
1       10.34  1.66  Male    No  Sun  Dinner  3  0.160542
2       21.01  3.50  Male    No  Sun  Dinner  3  0.166587
3       23.68  3.31  Male    No  Sun  Dinner  2  0.139780
4       24.59  3.61 Female    No  Sun  Dinner  4  0.146808
..     ...    ...     ...       ...     ...     ...       ...
239     29.03  5.92  Male    No  Sat  Dinner  3  0.203927
240     27.18  2.00 Female    Yes  Sat  Dinner  2  0.073584
241     22.67  2.00  Male    Yes  Sat  Dinner  2  0.088222
242     17.82  1.75  Male    No  Sat  Dinner  2  0.098204
243     18.78  3.00 Female    No  Thur  Dinner  2  0.159744

[244 rows x 8 columns]
```
WHERE

Filtering in SQL is done via a WHERE clause.

```sql
SELECT *
FROM tips
WHERE time = 'Dinner';
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```python
In [8]: tips[tips["total_bill"] > 10]
Out[8]:
   total_bill  tip     sex smoker day  time  size
0        16.99  1.01 Female  No  Sun Dinner  2
1        10.34  1.66   Male  No  Sun Dinner  3
2        21.01  3.50   Male  No  Sun Dinner  3
3        23.68  3.31   Male  No  Sun Dinner  2
4        24.59  3.61 Female  No  Sun Dinner  4
...       ...   ...     ...  ...    ...  ...  ...
239      29.03  5.92   Male  No  Sat Dinner  3
240      27.18  2.00 Female  Yes  Sat Dinner  2
241      22.67  2.00   Male  Yes  Sat Dinner  2
242      17.82  1.75   Male  No  Sat Dinner  2
243      18.78  3.00 Female  No  Thur Dinner  2
[227 rows x 7 columns]
```

The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

```python
In [9]: is_dinner = tips["time"] == "Dinner"
In [10]: is_dinner
Out[10]:
0  True
1  True
2  True
3  True
4  True
...  ...
239  True
240  True
241  True
242  True
243  True
Name: time, Length: 244, dtype: bool
```

```python
In [11]: is_dinner.value_counts()
Out[11]:
True    176
False    68
Name: time, dtype: int64
```

(continues on next page)
Tips of more than $5 at Dinner meals:

```sql
SELECT * 
FROM tips 
WHERE time = 'Dinner' AND tip > 5.00;
```

Tips by parties of at least 5 diners OR bill total was more than $45:

```sql
SELECT * 
FROM tips 
WHERE size >= 5 OR total_bill > 45;
```
NULL checking is done using the `notna()` and `isna()` methods.

```python
In [15]: frame = pd.DataFrame(
        ....: )

In [16]: frame
Out[16]:
   col1 col2
0    A    F
1    B  NaN
2  NaN    G
3    C    H
4    D    I
```

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```sql
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```python
In [17]: frame[frame["col2"].isna()]
Out[17]:
   col1 col2
0    B  NaN
```

Getting items where `col1` IS NOT NULL can be done with `notna()`.

```sql
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```python
In [18]: frame[frame["col1"].notna()]
Out[18]:
   col1 col2
0    D    I
```
GROUP BY

In pandas, SQL's GROUP BY operations are performed using the similarly named `groupby()` method. `groupby()` typically refers to a process where we'd like to split a dataset into groups, apply some function (typically aggregation), and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```sql
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female 87
Male 157
*/
```

The pandas equivalent would be:

```python
In [19]: tips.groupby("sex").size()
Out[19]:
sex
Female 87
Male 157
dtype: int64
```

Notice that in the pandas code we used `size()` and not `count()`. This is because `count()` applies the function to each column, returning the number of NOT NULL records within each.

```python
In [20]: tips.groupby("sex").count()
Out[20]:
          total_bill  tip  smoker  day  time  size
sex        Female  87   87    87    87   87
           Male  157  157  157  157  157
```

Alternatively, we could have applied the `count()` method to an individual column:

```python
In [21]: tips.groupby("sex")["total_bill"].count()
Out[21]:
sex
Female  87
Male  157
Name: total_bill, dtype: int64
```

Multiple functions can also be applied at once. For instance, say we'd like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.
```
SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
/*
Fri 2.734737 19
Sat 2.993103 87
Sun 3.255132 76
Thu 2.771452 62
*/
```

```
In [22]: tips.groupby("day").agg({"tip": np.mean, "day": np.size})
Out[22]:
    tip  day
      --- ----
   Fri 2.734737 19
  Sat 2.993103 87
  Sun 3.255132 76
  Thur 2.771452 62
```

Grouping by more than one column is done by passing a list of columns to the `groupby()` method.

```
SELECT smoker, day, COUNT(*), AVG(tip)
FROM tips
GROUP BY smoker, day;
/*
smoker  day
No    Fri 4  2.812500
      Sat 45 3.102889
      Sun 57 3.167895
      Thur 45 2.673778
Yes   Fri 15 2.714000
      Sat 42 2.875476
      Sun 19 3.516842
      Thur 17 3.030000
*/
```

```
In [23]: tips.groupby(["smoker", "day"]).agg({"tip": [np.size, np.mean]})
Out[23]:
    tip         size     mean
      ---          ----     -----
   No Fri 4  2.812500  2.812500
      Sat 45 3.102889  3.102889
      Sun 57 3.167895  3.167895
      Thur 45 2.673778  2.673778
Yes Fri 15 2.714000  2.714000
      Sat 42 2.875476  2.875476
      Sun 19 3.516842  3.516842
      Thur 17 3.030000  3.030000
```
JOIN

JOINs can be performed with `join()` or `merge()`. By default, `join()` will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

**Warning:** If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

```python
In [24]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})
In [25]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})
```

Assume we have two database tables of the same name and structure as our DataFrames.

Now let's go over the various types of JOINs.

**INNER JOIN**

```sql
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
```

```python
In [26]: pd.merge(df1, df2, on="key")
Out[26]:
key  value_x  value_y
0   B  -0.282863  1.212112
1   D  -1.135632 -0.173215
2   D  -1.135632  0.119209
```

`merge()` also offers parameters for cases when you’d like to join one DataFrame’s column with another DataFrame’s index.

```python
In [27]: indexed_df2 = df2.set_index("key")
In [28]: pd.merge(df1, indexed_df2, on="key", left_on="key", right_index=True)
Out[28]:
key  value_x  value_y
1   B  -0.282863  1.212112
3   D  -1.135632 -0.173215
3   D  -1.135632  0.119209
```
LEFT OUTER JOIN

Show all records from df1.

```
SELECT *
FROM df1
LEFT OUTER JOIN df2
    ON df1.key = df2.key;
```

```
In [29]: pd.merge(df1, df2, on="key", how="left")
Out[29]:
   key  value_x  value_y
0   A   0.469112     NaN
1   B  -0.282863  1.212112
2   C  -1.509059     NaN
3   D  -1.135632  -0.173215
4   D  -1.135632   0.119209
```

RIGHT JOIN

Show all records from df2.

```
SELECT *
FROM df1
RIGHT OUTER JOIN df2
    ON df1.key = df2.key;
```

```
In [30]: pd.merge(df1, df2, on="key", how="right")
Out[30]:
   key  value_x  value_y
0   B  -0.282863  1.212112
1   D  -1.135632 -0.173215
2   D  -1.135632  0.119209
3   E   NaN  -1.044236
```

FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

Show all records from both tables.

```
SELECT *
FROM df1
FULL OUTER JOIN df2
    ON df1.key = df2.key;
```

```
In [31]: pd.merge(df1, df2, on="key", how="outer")
Out[31]:
   key  value_x  value_y
   ... (continues on next page)
UNION

UNION ALL can be performed using `concat()`.

```python
In [32]: df1 = pd.DataFrame(
   ....:     {"city": ["Chicago", "San Francisco", "New York City"], "rank": range(1, 4)}
   ....: )
   ....:

In [33]: df2 = pd.DataFrame(
   ....:     {"city": ["Chicago", "Boston", "Los Angeles"], "rank": [1, 4, 5]}
   ....: )
   ....:
```

```sql
SELECT city, rank
FROM df1
UNION ALL
SELECT city, rank
FROM df2;
/*
   city    rank
Chicago   1
San Francisco  2
New York City  3
   Chicago   1
   Boston    4
Los Angeles   5
*/
```

```python
In [34]: pd.concat([df1, df2])
Out[34]:
   city    rank
0  Chicago   1
1  San Francisco   2
2  New York City   3
0  Chicago   1
1    Boston    4
2  Los Angeles   5
```

SQL's UNION is similar to UNION ALL, however UNION will remove duplicate rows.

```sql
SELECT city, rank
FROM df1
```
UNION
SELECT city, rank
FROM df2;
-- notice that there is only one Chicago record this time
/*
   city  rank
   Chicago 1
   San Francisco 2
   New York City 3
   Boston 4
   Los Angeles 5
*/

In pandas, you can use `concat()` in conjunction with `drop_duplicates()`.

```
In [35]: pd.concat([df1, df2]).drop_duplicates()
Out[35]:
   city   rank
0  Chicago 1
1  San Francisco 2
2    New York City 3
1    Boston 4
2    Los Angeles 5
```

LIMIT

```
SELECT * FROM tips
LIMIT 10;
```

```
In [36]: tips.head(10)
Out[36]:
   total_bill  tip  sex  smoker  day    time  size
0      16.99 1.01  Female     No  Sun  Dinner  2
1       10.34 1.66   Male     No  Sun  Dinner  3
2       21.01 3.50   Male     No  Sun  Dinner  3
3       23.68 3.31   Male     No  Sun  Dinner  2
4       24.59 3.61  Female     No  Sun  Dinner  4
5       25.29 4.71   Male     No  Sun  Dinner  4
6        8.77 2.00   Male     No  Sun  Dinner  2
7       26.88 3.12   Male     No  Sun  Dinner  4
8       15.04 1.96   Male     No  Sun  Dinner  2
9       14.78 3.23   Male     No  Sun  Dinner  2
```
pandas equivalents for some SQL analytic and aggregate functions

**Top n rows with offset**

```sql
-- MySQL
SELECT *
FROM tips
ORDER BY tip DESC
LIMIT 10 OFFSET 5;
```

```
In [37]: tips.nlargest(10 + 5, columns="tip").tail(10)
Out[37]:
    total_bill  tip  sex  smoker  day   time  size
183    23.17  6.50  Male    Yes   Sun  Dinner   4
214    28.17  6.50 Female   Yes   Sat  Dinner   3
  47    32.40  6.00  Male      No   Sun  Dinner   4
239    29.03  5.92  Male      No   Sat  Dinner   3
   88    24.71  5.85  Male      No  Thur  Lunch   2
181    23.33  5.65  Male    Yes   Sun  Dinner   2
  44    30.40  5.60  Male      No   Sun  Dinner   4
  52    34.81  5.20 Female     No   Sun  Dinner   4
  85    34.83  5.17 Female     No  Thur  Lunch   4
211    25.89  5.16  Male    Yes   Sat  Dinner   4
```

**Top n rows per group**

```sql
-- Oracle's ROW_NUMBER() analytic function
SELECT *
FROM (  
  SELECT  
    t.*,  
    ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn  
  FROM tips t  
)  
WHERE rn < 3  
ORDER BY day, rn;
```

```
In [38]: (
     ...
     ....:     tips.assign(  
     ....:       rn=tips.sort_values(["total_bill"], ascending=False)  
     ....:       .groupby(["day"])  
     ....:       .cumcount()  
     ....:       + 1  
     ....:     )  
     ....:     .query("rn < 3")  
     ....:     .sort_values(["day", "rn"])  
     ....: )  
     ....:
Out[38]:
    total_bill  tip  sex  smoker  day   time  size  rn
195    40.17  4.73  Male    Yes   Fri  Dinner   4   1
190    28.97  3.00  Male    Yes   Fri  Dinner   2   2
```

(continues on next page)
the same using `rank(method='first')` function

```
In [39]:
    (  
      ....:     tips.assign(  
      ....:         rnk=tips.groupby(['day'])['total_bill'].rank(  
      ....:             method='first', ascending=False
      ....:         )  
      ....:     ).query("rnk < 3")  
      ....:     .sort_values(['day', 'rnk'])
      ....:   )
   
   Out[39]:
    total_bill  tip  sex  smoker  day  time  size  rnk
   95  40.17  4.73  Male   Yes  Fri  Dinner  4 1.0
   90  28.97  3.00  Male   Yes  Fri  Dinner  2 2.0
  170  50.81 10.00  Male   Yes  Sat  Dinner  3 1.0
  212  48.33  9.00  Male    No  Sat  Dinner  4 2.0
  156  48.17  5.00  Male    No  Sun  Dinner  6 1.0
  182  45.35  3.50  Male   Yes  Sun  Dinner  3 2.0
  197  43.11  3.50  Female  Yes  Thur  Lunch  4 1.0
  142  41.19  5.00  Male    No  Thur  Lunch  5 2.0

-- Oracle's RANK() analytic function

SELECT * FROM (  
    SELECT  
      t.*,  
      RANK() OVER(PARTITION BY sex ORDER BY tip) AS rnk  
    FROM tips t  
    WHERE tip < 2  
)  
WHERE rnk < 3  
ORDER BY sex, rnk;
```

Let's find tips with (rank < 3) per gender group for (tips < 2). Notice that when using `rank(method='min')` function `rnk_min` remains the same for the same tip (as Oracle's RANK() function)

```
In [40]:
    (  
      ....:     tips[tips['tip'] < 2]
      ....:     .assign(rnk_min=tips.groupby(['sex'])['tip'].rank(method='min'))
      ....:     .query("rnk_min < 3")
      ....:     .sort_values(['sex', 'rnk_min'])
      ....:   )
   
   Out[40]:
```
In pandas we select the rows that should remain instead of deleting them:

In [42]: tips = tips.loc[tips["tip"] <= 9]

Comparison with spreadsheets

Since many potential pandas users have some familiarity with spreadsheet programs like Excel, this page is meant to provide some examples of how various spreadsheet operations would be performed using pandas. This page will use terminology and link to documentation for Excel, but much will be the same/similar in Google Sheets, LibreOffice Calc, Apple Numbers, and other Excel-compatible spreadsheet software.

If you’re new to pandas, you might want to first read through 10 Minutes to pandas to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

In [1]: import pandas as pd

In [2]: import numpy as np
Data structures

General terminology translation

<table>
<thead>
<tr>
<th>pandas</th>
<th>Excel</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame</td>
<td>worksheet</td>
</tr>
<tr>
<td>Series</td>
<td>column</td>
</tr>
<tr>
<td>Index</td>
<td>row headings</td>
</tr>
<tr>
<td>row</td>
<td>row</td>
</tr>
<tr>
<td>NaN</td>
<td>empty cell</td>
</tr>
</tbody>
</table>

**DataFrame**

A **DataFrame** in pandas is analogous to an Excel worksheet. While an Excel workbook can contain multiple worksheets, pandas **DataFrames** exist independently.

**Series**

A **Series** is the data structure that represents one column of a **DataFrame**. Working with a **Series** is analogous to referencing a column of a spreadsheet.

**Index**

Every **DataFrame** and **Series** has an **Index**, which are labels on the **rows** of the data. In pandas, if no index is specified, a **RangeIndex** is used by default (first row = 0, second row = 1, and so on), analogous to row headings/numbers in spreadsheets.

In pandas, indexes can be set to one (or multiple) unique values, which is like having a column that is used as the row identifier in a worksheet. Unlike most spreadsheets, these **Index** values can actually be used to reference the rows. (Note that this can be done in Excel with structured references.) For example, in spreadsheets, you would reference the first row as A1:Z1, while in pandas you could use `populations.loc['Chicago']`.

Index values are also persistent, so if you re-order the rows in a **DataFrame**, the label for a particular row don’t change. See the [indexing documentation](#) for much more on how to use an **Index** effectively.

**Copies vs. in place operations**

Most pandas operations return copies of the **Series/DataFrame**. To make the changes “stick”, you’ll need to either assign to a new variable:

```python
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```python
df = df.sort_values("col1")
```

**Note:** You will see an `inplace=True` keyword argument available for some methods:
df.sort_values("col1", inplace=True)

Its use is discouraged. More information.

Data input / output

Constructing a DataFrame from values

In a spreadsheet, values can be typed directly into cells.

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({"x": [1, 3, 5], ": y": [2, 4, 6]})
```

```
In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Both Excel and pandas can import data from various sources in various formats.

CSV

Let’s load and display the tips dataset from the pandas tests, which is a CSV file. In Excel, you would download and then open the CSV. In pandas, you pass the URL or local path of the CSV file to read_csv():

```
```

```
In [6]: tips = pd.read_csv(url)
```

```
In [7]: tips
Out[7]:
     total_bill  tip  sex smoker  day  time  size
0     16.99   1.01 Female       No  Sun Dinner 2
1     10.34   1.66    Male       No  Sun Dinner 3
2     21.01   3.50    Male       No  Sun Dinner 3
3     23.68   3.31    Male       No  Sun Dinner 2
4     24.59   3.61 Female       No  Sun Dinner 4
...     ...     ...     ...     ...     ...     ...
239   29.03   5.92    Male       No  Sat Dinner 3
```

(continues on next page)
Like Excel’s Text Import Wizard, `read_csv` can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```python
# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table("tips.csv", header=None)
```

### Excel files

Excel opens various Excel file formats by double-clicking them, or using the Open menu. In pandas, you use special methods for reading and writing from/to Excel files.

Let’s first create a new Excel file based on the `tips` dataframe in the above example:

```python
tips.to_excel("./tips.xlsx")
```

Should you wish to subsequently access the data in the `tips.xlsx` file, you can read it into your module using

```python
tips_df = pd.read_excel("./tips.xlsx", index_col=0)
```

You have just read in an Excel file using pandas!

### Limiting output

Spreadsheet programs will only show one screenful of data at a time and then allow you to scroll, so there isn’t really a need to limit output. In pandas, you’ll need to put a little more thought into controlling how your DataFrames are displayed.

By default, pandas will truncate output of large DataFrames to show the first and last rows. This can be overridden by changing the pandas options, or using `DataFrame.head()` or `DataFrame.tail()`.

```python
in [8]: tips.head(5)
Out[8]:
   total_bill  tip  sex  smoker  day  time  size
0     16.99  1.01 Female   No  Sun  Dinner   2
1     10.34  1.66   Male   No  Sun  Dinner   3
2     21.01  3.50   Male   No  Sun  Dinner   3
3     23.68  3.31   Male   No  Sun  Dinner   2
4     24.59  3.61 Female   No  Sun  Dinner   4
```
Exporting data

By default, desktop spreadsheet software will save to its respective file format (.xlsx, .ods, etc). You can, however, save to other file formats.

_pandas can create Excel files, CSV, or a number of other formats._

Data operations

Operations on columns

In spreadsheets, formulas are often created in individual cells and then dragged into other cells to compute them for other columns. In pandas, you’re able to do operations on whole columns directly.

pandas provides vectorized operations by specifying the individual Series in the DataFrame. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the DataFrame.

```
In [9]: tips["total_bill"] = tips["total_bill"] - 2
In [10]: tips["new_bill"] = tips["total_bill"] / 2
In [11]: tips
Out[11]:
   total_bill  tip  sex  smoker  day  time  size  new_bill
0       14.99  1.01  Female    No  Sun  Dinner     2      7.495
1        8.34  1.66    Male    No  Sun  Dinner     3      4.170
2       19.01  3.50    Male    No  Sun  Dinner     3      9.505
3       21.68  3.31    Male    No  Sun  Dinner     2     10.840
4       22.59  3.61  Female    No  Sun  Dinner     4     11.295
      ...    ...    ...    ...    ...    ...    ...    ...    ...
239    27.03  5.92    Male    No  Sat  Dinner     3     13.515
240    25.18  2.00  Female    Yes  Sat  Dinner     2     12.590
241    20.67  2.00    Male    Yes  Sat  Dinner     2     10.335
242    15.82  1.75    Male    No  Sat  Dinner     2      7.910
243    16.78  3.00  Female    No  Thur  Dinner     2      8.390
[244 rows x 8 columns]
```

In [12]: tips = tips.drop("new_bill", axis=1)

Note that we aren’t having to tell it to do that subtraction cell-by-cell — pandas handles that for us. See how to create new columns derived from existing columns.
Filtering

In Excel, filtering is done through a graphical menu.

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```python
In [13]: tips[tips["total_bill"] > 10]
Out[13]:
   total_bill  tip  sex smoker  day  time  size
1          10.00  1.00  Female     No  Sun  Dinner
2          12.50  2.54  Male      NO  Sat  Dinner
```

(continues on next page)
The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

```python
In [14]: is_dinner = tips["time"] == "Dinner"
In [15]: is_dinner
Out[15]:
0   True
1   True
2   True
3   True
4   True
...        
239  True
240  True
241  True
242  True
243  True
Name: time, Length: 244, dtype: bool

In [16]: is_dinner.value_counts()
Out[16]:
True    176
False    68
Name: time, dtype: int64

In [17]: tips[is_dinner]
Out[17]:
   total_bill  tip  sex  smoker day  time
0       14.99  1.01  Female  No  Sun  Dinner  2
1        8.34  1.66    Male  No  Sun  Dinner  3
2       19.01  3.50    Male  No  Sun  Dinner  3
3       21.68  3.31    Male  No  Sun  Dinner  2
4       22.59  3.61  Female  No  Sun  Dinner  4
5       23.29  4.71    Male  No  Sun  Dinner  4
...      ...     ...     ...    ...     ...  ...
239     27.03  5.92    Male  No  Sat  Dinner  3
240     25.18  2.00  Female  Yes  Sat  Dinner  2
241     20.67  2.00    Male  Yes  Sat  Dinner  2
242     15.82  1.75    Male  No  Sat  Dinner  2
243     16.78  3.00  Female  No  Thur  Dinner  2
```
If/then logic

Let’s say we want to make a bucket column with values of low and high, based on whether the total_bill is less or more than $10.

In spreadsheets, logical comparison can be done with conditional formulas. We’d use a formula of \( \text{=IF}(A2 < 10, \text{"low"}, \text{"high"}) \), dragged to all cells in a new bucket column.

The same operation in pandas can be accomplished using the where method from numpy.

The same operation in pandas can be accomplished using the where method from numpy.

In [18]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")

In [19]: tips
Out[19]:

[244 rows x 8 columns]
Date functionality

This section will refer to “dates”, but timestamps are handled similarly.

We can think of date functionality in two parts: parsing, and output. In spreadsheets, date values are generally parsed automatically, though there is a `DATEVALUE` function if you need it. In pandas, you need to explicitly convert plain text to datetime objects, either while reading from a CSV or once in a DataFrame.

Once parsed, spreadsheets display the dates in a default format, though the format can be changed. In pandas, you’ll generally want to keep dates as datetime objects while you’re doing calculations with them. Outputting parts of dates (such as the year) is done through date functions in spreadsheets, and datetime properties in pandas.

Given `date1` and `date2` in columns A and B of a spreadsheet, you might have these formulas:

<table>
<thead>
<tr>
<th>column</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1_year</td>
<td>=YEAR(A2)</td>
</tr>
<tr>
<td>date2_month</td>
<td>=MONTH(B2)</td>
</tr>
<tr>
<td>date1_next</td>
<td>=DATE(YEAR(A2),MONTH(A2)+1,1)</td>
</tr>
<tr>
<td>months_between</td>
<td>=DATEDIF(A2,B2,&quot;M&quot;)</td>
</tr>
</tbody>
</table>

The equivalent pandas operations are shown below.

```
In [20]: tips["date1"] = pd.Timestamp("2013-01-15")

In [21]: tips["date2"] = pd.Timestamp("2015-02-15")

In [22]: tips["date1_year"] = tips["date1"].dt.year

In [23]: tips["date2_month"] = tips["date2"].dt.month

In [24]: tips["date1_next"] = tips["date1"] + pd.offsets.MonthBegin()

In [25]: tips["months_between"] = tips["date2"].dt.to_period("M") - tips["date1"].dt.to_period("M")

In [26]: tips["date1", "date2", "date1_year", "date2_month", "date1_next", "months_between"]

Out[26]:
```
```
See *Time series / date functionality* for more details.

**Selection of columns**

In spreadsheets, you can select columns you want by:

- Hiding columns
- Deleting columns
- Referencing a range from one worksheet into another

Since spreadsheet columns are typically named in a header row, renaming a column is simply a matter of changing the text in that first cell.

The same operations are expressed in pandas below.

**: Keep certain columns**

```python
In [27]: tips[['sex', 'total_bill', 'tip']]
Out[27]:
          sex  total_bill  tip
       0  Female  14.99   1.01
       1    Male   8.34   1.66
       2    Male  19.01   3.50
       3    Male  21.68   3.31
       4  Female  22.59   3.61
          ...        ...  ...
     239    Male  27.03   5.92
     240  Female  25.18   2.00
     241    Male  20.67   2.00
     242    Male  15.82   1.75
     243  Female  16.78   3.00
```

**Drop a column**

```python
In [28]: tips.drop("sex", axis=1)
Out[28]:
          total_bill  tip  smoker  day  time  size
       0    14.99  1.01   No  Sun  Dinner   2
       1     8.34  1.66   No  Sun  Dinner   3
       2    19.01  3.50   No  Sun  Dinner   3
       3    21.68  3.31   No  Sun  Dinner   2
       4    22.59  3.61   No  Sun  Dinner   4
```
.. |239| 27.03 | 5.92 | No | Sat | Dinner | 3  
|240| 25.18 | 2.00 | Yes | Sat | Dinner | 2  
|241| 20.67 | 2.00 | Yes | Sat | Dinner | 2  
|242| 15.82 | 1.75 | No  | Sat | Dinner | 2  
|243| 16.78 | 3.00 | No  | Thur| Dinner | 2  

[244 rows x 6 columns]

## Rename a column

```python
In [29]: tips.rename(columns={"total_bill": "total_bill_2"})
```

```
Out[29]:
<table>
<thead>
<tr>
<th>total_bill_2</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14.99</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>8.34</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>19.01</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>21.68</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>22.59</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>239</td>
<td>27.03</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>240</td>
<td>25.18</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>241</td>
<td>20.67</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>242</td>
<td>15.82</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>243</td>
<td>16.78</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Dinner</td>
<td>2</td>
</tr>
</tbody>
</table>
```

[244 rows x 7 columns]

## Sorting by values

Sorting in spreadsheets is accomplished via the sort dialog.
pandas has a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```python
In [30]: tips = tips.sort_values(["sex", "total_bill"])

In [31]: tips
```

```
Out[31]:
    total_bill     tip     sex  smoker  day    time  size
  67     1.07  1.00 Female    Yes   Sat  Dinner   1
  92     3.75  1.00 Female    Yes   Fri  Dinner   2
 111     5.25  1.00 Female     No   Sat  Dinner   1
 145     6.35  1.50 Female     No  Thur  Lunch   2
 135     6.51  1.25 Female     No  Thur  Lunch   2
  85     6.08  1.50  Female     No  Thur  Lunch   2
 182    43.35  3.50    Male    Yes   Sun  Dinner   3
 156    46.17  5.00    Male     No  Sun  Dinner   6
  59    46.27  6.73    Male     No  Sat  Dinner   4
 212    46.33  9.00    Male     No  Sat  Dinner   4
 170    48.81 10.00    Male    Yes   Sat  Dinner   3

[244 rows x 7 columns]
```

**String processing**

**Finding length of string**

In spreadsheets, the number of characters in text can be found with the `LEN` function. This can be used with the `TRIM` function to remove extra whitespace.

```
=LEN(TRIM(A2))
```

You can find the length of a character string with `Series.str.len()`. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.
In [32]: tips["time"].str.len()
Out[32]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>6</td>
</tr>
<tr>
<td>92</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>6</td>
</tr>
<tr>
<td>145</td>
<td>5</td>
</tr>
<tr>
<td>135</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>..</td>
</tr>
<tr>
<td>182</td>
<td>6</td>
</tr>
<tr>
<td>156</td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td>6</td>
</tr>
<tr>
<td>212</td>
<td>6</td>
</tr>
<tr>
<td>170</td>
<td>6</td>
</tr>
</tbody>
</table>

Name: time, Length: 244, dtype: int64

In [33]: tips["time"].str.rstrip().str.len()
Out[33]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>6</td>
</tr>
<tr>
<td>92</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>6</td>
</tr>
<tr>
<td>145</td>
<td>5</td>
</tr>
<tr>
<td>135</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>..</td>
</tr>
<tr>
<td>182</td>
<td>6</td>
</tr>
<tr>
<td>156</td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td>6</td>
</tr>
<tr>
<td>212</td>
<td>6</td>
</tr>
<tr>
<td>170</td>
<td>6</td>
</tr>
</tbody>
</table>

Name: time, Length: 244, dtype: int64

Note this will still include multiple spaces within the string, so isn’t 100% equivalent.

**Finding position of substring**

The **FIND** spreadsheet function returns the position of a substring, with the first character being 1.
You can find the position of a character in a column of strings with the `Series.str.find()` method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns -1. Keep in mind that Python indexes are zero-based.

```
In [34]: tips["sex"].str.find("ale")
Out[34]:
67    3
92    3
111   3
145   3
135   3
   ...  
182   1
156   1
  59   1
212   1
170   1
Name: sex, Length: 244, dtype: int64
```

**Extracting substring by position**

Spreadsheets have a MID formula for extracting a substring from a given position. To get the first character:

```
=MID(A2,1,1)
```

With pandas you can use [] notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [35]: tips["sex"].str[0:1]
Out[35]:
67    F
92    F
```

(continues on next page)
Extracting nth word

In Excel, you might use the Text to Columns Wizard for splitting text and retrieving a specific column. (Note it’s possible to do so through a formula as well.)

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

```python
In [36]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})

In [37]: firstlast["First_Name"] = firstlast["String"]).str.split(" ", expand=True)[0]

In [38]: firstlast["Last_Name"] = firstlast["String"]).str.rsplit(" ", expand=True)[1]

In [39]: firstlast
Out[39]:
     String First_Name Last_Name
0     John Smith       John       Smith
1      Jane Cook       Jane        Cook
```

Changing case

Spreadsheets provide UPPER, LOWER, and PROPER functions for converting text to upper, lower, and title case, respectively.

The equivalent pandas methods are `Series.str.upper()`, `Series.str.lower()`, and `Series.str.title()`.

```python
In [40]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})

In [41]: firstlast["upper"] = firstlast["string"]).str.upper()

In [42]: firstlast["lower"] = firstlast["string"]).str.lower()

In [43]: firstlast["title"] = firstlast["string"]).str.title()

In [44]: firstlast
Out[44]:
     string   upper     lower     title
0      John Smith  JOHN SMITH  john smith   John Smith
1        Jane Cook   JANE COOK    jane cook     Jane Cook
```
Merging

The following tables will be used in the merge examples:

```python
In [45]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})
Out[46]:
key  value
0    A  0.469112
1    B -0.282863
2    C -1.509059
3    D -1.135632
```

```python
In [47]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})
Out[48]:
key  value
0    B  1.212112
1    D -0.173215
2    D  0.119209
3    E -1.044236
```

In Excel, there are merging of tables can be done through a VLOOKUP.

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pandas DataFrames have a `merge()` method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

```python
In [49]: inner_join = df1.merge(df2, on="key", how="inner")
Out[50]:
key  value_x  value_y
0    B -0.282863  1.212112
1    D -1.135632 -0.173215
2    D -1.135632  0.119209
```

(continues on next page)
In [51]: left_join = df1.merge(df2, on=["key"], how="left")

In [52]: left_join
Out[52]:
   key  value_x  value_y  
0   A   0.469112    NaN  
1   B  -0.282863  1.212112  
2   C  -1.509059    NaN  
3   D  -1.135632 -0.173215  
4   D  -1.135632   0.119209  

In [53]: right_join = df1.merge(df2, on=["key"], how="right")

In [54]: right_join
Out[54]:
   key  value_x  value_y  
0   B  -0.282863  1.212112  
1   D  -1.135632 -0.173215  
2   D  -1.135632   0.119209  
3   E    NaN  -1.044236  

In [55]: outer_join = df1.merge(df2, on=["key"], how="outer")

In [56]: outer_join
Out[56]:
   key  value_x  value_y  
0   A   0.469112    NaN  
1   B  -0.282863  1.212112  
2   C  -1.509059    NaN  
3   D  -1.135632 -0.173215  
4   D  -1.135632   0.119209  
5   E    NaN  -1.044236  

merge has a number of advantages over VLOOKUP:

   • The lookup value doesn’t need to be the first column of the lookup table
   • If multiple rows are matched, there will be one row for each match, instead of just the first
   • It will include all columns from the lookup table, instead of just a single specified column
   • It supports more complex join operations

Other considerations

Fill Handle

Create a series of numbers following a set pattern in a certain set of cells. In a spreadsheet, this would be done by
shift+drag after entering the first number or by entering the first two or three values and then dragging.

This can be achieved by creating a series and assigning it to the desired cells.
In [57]: df = pd.DataFrame({'AAA': [1] * 8, 'BBB': list(range(0, 8)))

In [58]: df
Out[58]:
   AAA  BBB
0   1    0
1   1    1
2   1    2
3   1    3
4   1    4
5   1    5
6   1    6
7   1    7

In [59]: series = list(range(1, 5))

In [60]: series
Out[60]: [1, 2, 3, 4]

In [61]: df.loc[2:5, "AAA"] = series

In [62]: df
Out[62]:
   AAA  BBB
0   1    0
1   1    1
2   1    2
3   2    3
4   3    4
5   4    5
6   1    6
7   1    7

Drop Duplicates

Excel has built-in functionality for removing duplicate values. This is supported in pandas via `drop_duplicates()`.

In [63]: df = pd.DataFrame(
    ...:     {
    ...:         "class": ["A", "A", "A", "B", "C", "D"],
    ...:         "student_count": [42, 35, 42, 50, 47, 45],
    ...:         "all_pass": ["Yes", "Yes", "Yes", "No", "No", "Yes"],
    ...:     })
In [64]: df.drop_duplicates()
Out[64]:
     class  student_count  all_pass
0     A          42       Yes
1     A          35       Yes

(continues on next page)
In [65]: df.drop_duplicates(["class", "student_count"])
Out[65]:
   class  student_count  all_pass
0       A           42     Yes
1       A           35     Yes
2       B           50      No
3       C           47      No
4       D           45     Yes

Pivot Tables

PivotTables from spreadsheets can be replicated in pandas through Reshaping and pivot tables. Using the tips dataset again, let’s find the average gratuity by size of the party and sex of the server.

In Excel, we use the following configuration for the PivotTable:

The equivalent in pandas:

In [66]: pd.pivot_table(
       ....:     tips, values="tip", index="size", columns="sex", aggfunc=np.average
       ....: )
Out[66]:
   sex     Female  Male
size    
1      1.276667  1.920000
2      2.528482  2.614184
3      3.250000  3.476667
4      4.021111  4.172143

Adding a row

Assuming we are using a `RangeIndex` (numbered 0, 1, etc.), we can use `concat()` to add a row to the bottom of a DataFrame.

```python
In [67]: df
Out[67]:
   class  student_count  all_pass
0      A            42      Yes
1      A            35      Yes
2      A            42      Yes
3      B            50      No
4      C            47      No
5      D            45      Yes
```

```python
In [68]: new_row = pd.DataFrame([["E", 51, True]],
                           columns=["class", "student_count", "all_pass"])
....:
In [69]: pd.concat([df, new_row])
Out[69]:
   class  student_count  all_pass
0      A            42      Yes
1      A            35      Yes
2      A            42      Yes
3      B            50      No
4      C            47      No
5      D            45      Yes
6      E            51      True
```

Find and Replace

Excel's Find dialog takes you to cells that match, one by one. In pandas, this operation is generally done for an entire column or `DataFrame` at once through `conditional expressions`.

```python
In [70]: tips
Out[70]:
   total_bill  tip  sex  smoker  day  time  size
67  6.35  1.25  Female  No  Thur  Lunch  2
92  3.75  1.00  Female  Yes  Fri  Dinner  2
111 5.25  1.00  Female  No  Sat  Dinner  1
145 6.35  1.50  Female  No  Thur  Lunch  2
135 6.51  1.25  Female  No  Thur  Lunch  2
... ... ... ... ... ... ...
182 43.35  3.50  Male  Yes  Sun  Dinner  3
156 46.17  5.00  Male  No  Sun  Dinner  6
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

59 46.27 6.73 Male No Sat Dinner 4
212 46.33 9.00 Male No Sat Dinner 4
170 48.81 10.00 Male Yes Sat Dinner 3

[244 rows x 7 columns]

In [71]: tips == "Sun"
Out[71]:

In [72]: tips["day"].str.contains("S")

Out[72]:

In [73]: tips.replace("Thu", "Thursday")

pandas' replace() is comparable to Excel's Replace All.
Comparison with SAS

For potential users coming from SAS this page is meant to demonstrate how different SAS operations would be performed in pandas.

If you’re new to pandas, you might want to first read through 10 Minutes to pandas to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```python
In [1]: import pandas as pd
In [2]: import numpy as np
```

Data structures

General terminology translation

<table>
<thead>
<tr>
<th>pandas</th>
<th>SAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame</td>
<td>data set</td>
</tr>
<tr>
<td>column</td>
<td>variable</td>
</tr>
<tr>
<td>row</td>
<td>observation</td>
</tr>
<tr>
<td>groupby</td>
<td>BY-group</td>
</tr>
<tr>
<td>NaN</td>
<td>.</td>
</tr>
</tbody>
</table>

DataFrame

A DataFrame in pandas is analogous to a SAS data set - a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set using SAS’s DATA step, can also be accomplished in pandas.

Series

A Series is the data structure that represents one column of a DataFrame. SAS doesn’t have a separate data structure for a single column, but in general, working with a Series is analogous to referencing a column in the DATA step.
Index

Every DataFrame and Series has an Index - which are labels on the rows of the data. SAS does not have an exactly analogous concept. A data set’s rows are essentially unlabeled, other than an implicit integer index that can be accessed during the DATA step (_N_).

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the Index and just treat the DataFrame as a collection of columns. Please see the indexing documentation for much more on how to use an Index effectively.

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes “stick”, you’ll need to either assign to a new variable:

```python
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```python
df = df.sort_values("col1")
```

Note: You will see an inplace=True keyword argument available for some methods:

```python
df.sort_values("col1", inplace=True)
```

Its use is discouraged. More information.

Data input / output

Constructing a DataFrame from values

A SAS data set can be built from specified values by placing the data after a datalines statement and specifying the column names.

```sas
data df;
   input x y;
datalines;
   1 2
   3 4
   5 6
;run;
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```python
In [1]: df = pd.DataFrame({"x": [1, 3, 5], "y": [2, 4, 6]})
In [2]: df
```

(continues on next page)
Reading external data

Like SAS, pandas provides utilities for reading in data from many formats. The tips dataset, found within the pandas tests (csv) will be used in many of the following examples.

SAS provides PROC IMPORT to read csv data into a data set.

```sql
proc import datafile='tips.csv' dbms=csv out=tips replace;
   getnames=yes;
run;
```

The pandas method is `read_csv()`, which works similarly.

```python
In [3]: url = (  
          ...:   "https://raw.github.com/pandas-dev/"  
          ...:   "pandas/main/pandas/tests/io/data/csv/tips.csv"  
          ...: )  
          ...: 
In [4]: tips = pd.read_csv(url)

In [5]: tips
Out[5]:
   total_bill  tip  sex  smoker  day  time  size
0       16.99  1.01 Female   No   Sun  Dinner   2
1       10.34  1.66     Male   No   Sun  Dinner   3
2       21.01  3.50     Male   No   Sun  Dinner   3
3       23.68  3.31     Male   No   Sun  Dinner   2
4       24.59  3.61 Female   No   Sun  Dinner   4
   ...    ...    ...    ...    ...    ...    ...
239    29.03  5.92     Male   No   Sat  Dinner   3
240    27.18  2.00 Female   Yes   Sat  Dinner   2
241    22.67  2.00     Male   Yes   Sat  Dinner   2
242    17.82  1.75     Male   No   Sat  Dinner   2
243    18.78  3.00 Female   No   Thur  Dinner   2
[244 rows x 7 columns]
```

Like PROC IMPORT, read_csv can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```python
tips = pd.read_csv("tips.csv", sep="\t", header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
```
In addition to text/csv, pandas supports a variety of other data formats such as Excel, HDF5, and SQL databases. These are all read via a `pd.read_*` function. See the IO documentation for more details.

**Limiting output**

By default, pandas will truncate output of large DataFrames to show the first and last rows. This can be overridden by changing the pandas options, or using `DataFrame.head()` or `DataFrame.tail()`.

```
In [1]: tips.head(5)
Out[1]:
     total_bill  tip   sex  smoker day    time  size
0   16.990000  1.01 Female   No  Sun  Dinner    2
1   10.340000  1.66   Male   No  Sun  Dinner    3
2   21.010000  3.50   Male   No  Sun  Dinner    3
3   23.680000  3.31   Male   No  Sun  Dinner    2
4   24.590000  3.61 Female   No  Sun  Dinner    4
```

The equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

**Exporting data**

The inverse of `PROC IMPORT` in SAS is `PROC EXPORT`

```
proc export data=tips outfile='tips2.csv' dbms=csv;
run;
```

Similarly in pandas, the opposite of `read_csv` is `to_csv()`, and other data formats follow a similar api.

```
tips.to_csv("tips2.csv")
```

**Data operations**

**Operations on columns**

In the DATA step, arbitrary math expressions can be used on new or existing columns.

```
data tips;
   set tips;
   total_bill = total_bill - 2;
   new_bill = total_bill / 2;
run;
```

pandas provides vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`. 

112 Chapter 1. Getting started
In [1]: tips["total_bill"] = tips["total_bill"] - 2

In [2]: tips["new_bill"] = tips["total_bill"] / 2

In [3]: tips

Out[3]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>new_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>7.495</td>
</tr>
<tr>
<td>8.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>4.170</td>
</tr>
<tr>
<td>19.01</td>
<td>3.50</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>9.505</td>
</tr>
<tr>
<td>21.68</td>
<td>3.31</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>10.840</td>
</tr>
<tr>
<td>22.59</td>
<td>3.61</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
<td>11.295</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>27.03</td>
<td>5.92</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
<td>13.515</td>
</tr>
<tr>
<td>25.18</td>
<td>2.00</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>12.590</td>
</tr>
<tr>
<td>20.67</td>
<td>2.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>10.335</td>
</tr>
<tr>
<td>15.82</td>
<td>1.75</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>7.910</td>
</tr>
<tr>
<td>16.78</td>
<td>3.00</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Dinner</td>
<td>2</td>
<td>8.390</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

[244 rows x 8 columns]

In [4]: tips = tips.drop("new_bill", axis=1)

Filtering

Filtering in SAS is done with an if or where statement, on one or more columns.

data tips;
    set tips;
    if total_bill > 10;
run;

data tips;
    set tips;
    where total_bill > 10;
    /* equivalent in this case - where happens before the DATA step begins and can also be used in PROC statements */
run;

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

In [1]: tips[tips["total_bill"] > 10]

Out[1]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>19.01</td>
<td>3.50</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>21.68</td>
<td>3.31</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>22.59</td>
<td>3.61</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>23.29</td>
<td>4.71</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>27.03</td>
<td>5.92</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>25.18</td>
<td>2.00</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
</tr>
</tbody>
</table>

(continues on next page)
The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

```python
In [1]: is_dinner = tips["time"] == "Dinner"

In [2]: is_dinner
Out[2]:
0   True
1   True
2   True
3   True
4   True
   ... 
239  True
240  True
241  True
242  True
243  True
Name: time, Length: 244, dtype: bool

In [3]: is_dinner.value_counts()
Out[3]:
True    176
False    68
Name: time, dtype: int64

In [4]: tips[is_dinner]
Out[4]:
   total_bill  tip sex  smoker day  time  size
0    14.99  1.01  Female  No  Sun  Dinner  2
1     8.34  1.66    Male  No  Sun  Dinner  3
2    19.01  3.50    Male  No  Sun  Dinner  3
3    21.68  3.31    Male  No  Sun  Dinner  2
4    22.59  3.61  Female  No  Sun  Dinner  4
   ...   ...   ...   ...   ...   ...   ...
239  27.03  5.92    Male  No  Sat  Dinner  3
240  25.18  2.00  Female  Yes  Sat  Dinner  2
241  20.67  2.00    Male  Yes  Sat  Dinner  2
242  15.82  1.75    Male  No  Sat  Dinner  2
243  16.78  3.00  Female  No  Thur  Dinner  2
[176 rows x 7 columns]
If/then logic

In SAS, if/then logic can be used to create new columns.

```sas
data tips;
  set tips;
  format bucket $4.;
  if total_bill < 10 then bucket = 'low';
  else bucket = 'high';
run;
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```python
In [1]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")
```

```
In [2]: tips
Out[2]:
   total_bill  tip     sex  smoker  day  time  size  bucket
     0    14.99  1.01 Female   No  Sun  Dinner   2  high
     1     8.34  1.66   Male   No  Sun  Dinner   3   low
     2    19.01  3.50   Male   No  Sun  Dinner   3   high
     3    21.68  3.31   Male   No  Sun  Dinner   2  high
     4    22.59  3.61 Female   No  Sun  Dinner   4  high
    ..    ...    ...     ...     ...     ...     ...  ...
  239    27.03  5.92   Male   No  Sat  Dinner   3  high
  240    25.18  2.00 Female   Yes  Sat  Dinner   2  high
  241    20.67  2.00   Male   Yes  Sat  Dinner   2  high
  242    15.82  1.75   Male   No  Sat  Dinner   2  high
  243    16.78  3.00 Female   No  Thur  Dinner   2  high
[244 rows x 8 columns]
```

Date functionality

SAS provides a variety of functions to do operations on date/datetime columns.

```sas
data tips;
  set tips;
  format date1 date2 date1_plusmonth mmddyy10.;
  date1 = mdy(1, 15, 2013);
  date2 = mdy(2, 15, 2015);
  date1_year = year(date1);
  date2_month = month(date2);
  * shift date to beginning of next interval;
  datel_next = intnx('MONTH', datel, 1);
  * count intervals between dates;
  months_between = intck('MONTH', datel, date2);
run;
```

The equivalent pandas operations are shown below. In addition to these functions pandas supports other Time Series features not available in Base SAS (such as resampling and custom offsets) - see the `timeseries documentation` for more details.
### Selection of columns

SAS provides keywords in the **DATA** step to select, drop, and rename columns.

```sas
data tips;
   set tips;
   keep sex total_bill tip;
run;

data tips;
   set tips;
   drop sex;
run;

data tips;
```

(continues on next page)
The same operations are expressed in pandas below.

**Keep certain columns**

```python
In [1]: tips["sex", "total_bill", "tip"]
Out[1]:
   sex  total_bill  tip
 0  Female    14.99   1.01
 1   Male      8.34   1.66
 2   Male     19.01   3.50
 3   Male     21.68   3.31
 4  Female    22.59   3.61
 ..     ...     ... ...
239  Male    27.03   5.92
240 Female    25.18   2.00
241   Male    20.67   2.00
242   Male    15.82   1.75
243 Female    16.78   3.00
[244 rows x 3 columns]
```

**Drop a column**

```python
In [2]: tips.drop("sex", axis=1)
Out[2]:
   total_bill  tip  smoker  day  time  size
 0     14.99   1.01  No  Sun  Dinner    2
 1      8.34   1.66  No  Sun  Dinner    3
 2     19.01   3.50  No  Sun  Dinner    3
 3     21.68   3.31  No  Sun  Dinner    2
 4     22.59   3.61  No  Sun  Dinner    4
 ..     ...     ...     ...     ...     ...     ...
239    27.03   5.92  No  Sat  Dinner    3
240    25.18   2.00  Yes  Sat  Dinner    2
241    20.67   2.00  Yes  Sat  Dinner    2
242    15.82   1.75  No  Sat  Dinner    2
243    16.78   3.00  No  Thur  Dinner    2
[244 rows x 6 columns]
```
Rename a column

```python
In [1]: tips.rename(columns={"total_bill": "total_bill_2"})
Out[1]:

         total_bill_2  tip  sex  smoker  day     time  size
0       14.99    1.01 Female  No  Sun     Dinner   2
1       8.34    1.66  Male   No  Sun     Dinner   3
2      19.01    3.50  Male   No  Sun     Dinner   3
3      21.68    3.31  Male   No  Sun     Dinner   2
4      22.59    3.61 Female  No  Sun     Dinner   4
   ...     ...     ...    ...    ...       ...   ...
239    27.03    5.92  Male   No  Sat     Dinner   3
240    25.18    2.00 Female  Yes  Sat     Dinner   2
241    20.67    2.00  Male   Yes  Sat     Dinner   2
242    15.82    1.75  Male   No  Sat     Dinner   2
243    16.78    3.00 Female  No  Thur    Dinner   2
[244 rows x 7 columns]
```

Sorting by values

Sorting in SAS is accomplished via PROC_SORT

```sas
proc sort data=tips;
    by sex total_bill;
run;
```

pandas has a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```python
In [1]: tips = tips.sort_values(["sex", "total_bill"])
In [2]: tips
Out[2]:

         total_bill  tip  sex  smoker  day     time  size
67     1.07    1.00 Female  Yes  Sat     Dinner   1
92     3.75    1.00 Female  Yes  Fri     Dinner   2
111    5.25    1.00 Female  No  Sat     Dinner   1
145    6.35    1.50 Female  No  Thur    Lunch    2
135    6.51    1.25 Female  No  Thur    Lunch    2
   ...     ...     ...    ...    ...       ...   ...
182    43.35    3.50  Male   Yes  Sun     Dinner   3
156    46.17    5.00  Male   No  Sun     Dinner   6
59     46.27    6.73  Male   No  Sat     Dinner   4
212    46.33    9.00  Male   No  Sat     Dinner   4
170    48.81   10.00  Male   Yes  Sat     Dinner   3
[244 rows x 7 columns]
```
String processing

Finding length of string

SAS determines the length of a character string with the `LENGTHN` and `LENGTHC` functions. `LENGTHN` excludes trailing blanks and `LENGTHC` includes trailing blanks.

```plaintext
data _null_;  
set tips;  
put(LENGTHN(time));  
put(LENGTHC(time));  
run;
```

You can find the length of a character string with `Series.str.len()`. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```plaintext
In [1]: tips["time"].str.len()  
Out[1]:  
   67   6  
   92   6  
  111   6  
  145   5  
  135   5  
   ..  
  182   6  
  156   6  
   59   6  
  212   6  
  170   6  
Name: time, Length: 244, dtype: int64

In [2]: tips["time"].str.rstrip().str.len()  
Out[2]:  
   67   6  
   92   6  
  111   6  
  145   5  
  135   5  
   ..  
  182   6  
  156   6  
   59   6  
  212   6  
  170   6  
Name: time, Length: 244, dtype: int64
```
Finding position of substring

SAS determines the position of a character in a string with the `FINDW` function. `FINDW` takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```sas
data _null_;
set tips;
put(FINDW(sex,'ale'));
run;
```

You can find the position of a character in a column of strings with the `Series.str.find()` method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns -1. Keep in mind that Python indexes are zero-based.

```
In [1]: tips["sex"][str.find("ale")]
Out[1]:
67   3
92   3
111  3
145  3
135  3
... 
182  1
156  1
59   1
212  1
170  1
Name: sex, Length: 244, dtype: int64
```

Extracting substring by position

SAS extracts a substring from a string based on its position with the `SUBSTR` function.

```sas
data _null_;
set tips;
put(substr(sex,1,1));
run;
```

With pandas you can use [ ] notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [1]: tips["sex"][0:1]
Out[1]:
67   F
92   F
111  F
145  F
135  F
... 
182  M
156  M
59   M
```

(continues on next page)
Extracting nth word

The SAS \texttt{SCAN} function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

\begin{verbatim}
data firstlast;
  input String $60.;
  First_Name = scan(string, 1);
  Last_Name = scan(string, -1);
datalines2;
John Smith;
Jane Cook;
;;;
run;
\end{verbatim}

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

\begin{verbatim}
In [1]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})
In [2]: firstlast["First_Name"] = firstlast["String"].str.split(" ", expand=True)[0]
In [3]: firstlast["Last_Name"] = firstlast["String"].str.rsplit(" ", expand=True)[1]
In [4]: firstlast
Out[4]:
    String  First_Name  Last_Name
   0 John Smith       John    Smith
   1  Jane Cook       Jane     Cook
\end{verbatim}

Changing case

The SAS \texttt{UPCASE LOWCASE} and \texttt{PROPCASE} functions change the case of the argument.

\begin{verbatim}
data firstlast;
  input String $60.;
  string_up = UPCASE(string);
  string_low = LOWCASE(string);
  string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
;;;
run;
\end{verbatim}

The equivalent pandas methods are \texttt{Series.str.upper()}, \texttt{Series.str.lower()}, and \texttt{Series.str.title()}. 

1.4. Tutorials
In [1]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})

In [2]: firstlast["upper"] = firstlast["string"].str.upper()

In [3]: firstlast["lower"] = firstlast["string"].str.lower()

In [4]: firstlast["title"] = firstlast["string"].str.title()

In [5]: firstlast

Out[5]:

<table>
<thead>
<tr>
<th></th>
<th>string</th>
<th>upper</th>
<th>lower</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>John Smith</td>
<td>JOHN SMITH</td>
<td>john smith</td>
<td>John Smith</td>
</tr>
<tr>
<td>1</td>
<td>Jane Cook</td>
<td>JANE COOK</td>
<td>jane cook</td>
<td>Jane Cook</td>
</tr>
</tbody>
</table>

Merging

The following tables will be used in the merge examples:

In [1]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [2]: df1

Out[2]:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.469112</td>
</tr>
<tr>
<td>1</td>
<td>-0.282863</td>
</tr>
<tr>
<td>2</td>
<td>-1.509059</td>
</tr>
<tr>
<td>3</td>
<td>-1.135632</td>
</tr>
</tbody>
</table>

In [3]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

In [4]: df2

Out[4]:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.212112</td>
</tr>
<tr>
<td>1</td>
<td>-0.173215</td>
</tr>
<tr>
<td>2</td>
<td>0.119209</td>
</tr>
<tr>
<td>3</td>
<td>-1.044236</td>
</tr>
</tbody>
</table>

In SAS, data must be explicitly sorted before merging. Different types of joins are accomplished using the in= dummy variables to track whether a match was found in one or both input frames.

```plaintext
proc sort data=df1;
  by key;
run;

proc sort data=df2;
  by key;
run;

data left_join inner_join right_join outer_join;
  merge df1(in=a) df2(in=b);
```

(continues on next page)
if a and b then output inner_join;
if a then output left_join;
if b then output right_join;
if a or b then output outer_join;
run;

pandas DataFrames have a `merge()` method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the how keyword.

In [1]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [2]: inner_join
Out[2]:
key  value_x  value_y
θ  B  0.282863  1.212112
1  D -1.135632  0.119209
2  D -1.135632  0.119209

In [3]: left_join = df1.merge(df2, on=['key'], how='left')

In [4]: left_join
Out[4]:
key  value_x  value_y
θ  A  0.469112   NaN
1  B -0.282863  1.212112
2  C -1.509059   NaN
3  D -1.135632  0.119209
4  D -1.135632  0.119209
5  E   NaN  -1.044236

In [5]: right_join = df1.merge(df2, on=['key'], how='right')

In [6]: right_join
Out[6]:
key  value_x  value_y
θ  B -0.282863  1.212112
1  D -1.135632  0.119209
2  D -1.135632  0.119209
3  E   NaN  -1.044236
4  E   NaN  -1.044236

In [7]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [8]: outer_join
Out[8]:
key  value_x  value_y
θ  A  0.469112   NaN
1  B -0.282863  1.212112
2  C -1.509059   NaN
3  D -1.135632  0.119209
4  D -1.135632  0.119209
5  E   NaN  -1.044236

1.4. Tutorials
Missing data

Both pandas and SAS have a representation for missing data.
pandas represents missing data with the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [1]: outer_join
Out[1]:
   key  value_x  value_y
0   A  0.469112   NaN
1   B -0.282863  1.212112
2   C -1.509059   NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E   NaN  -1.044236

In [2]: outer_join["value_x"] + outer_join["value_y"]
Out[2]:
   NaN
   0.929249
   NaN
   NaN
   -1.308847
   -1.016424
   NaN
dtype: float64

In [3]: outer_join["value_x"].sum()
Out[3]: -3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in SAS you could do this to filter missing values.

```
data outer_join_nulls;
   set outer_join;
   if value_x = .;
run;

data outer_join_no_nulls;
   set outer_join;
   if value_x ^= .;
run;
```

In pandas, `Series.isna()` and `Series.notna()` can be used to filter the rows.

```
In [1]: outer_join[outer_join["value_x"].isna()]
Out[1]:
   key  value_x  value_y
5   E   NaN  -1.044236

In [2]: outer_join[outer_join["value_x"].notna()]
Out[2]:
   key  value_x  value_y
0   A  0.469112   NaN
```

(continues on next page)
pandas provides a variety of methods to work with missing data. Here are some examples:

**Drop rows with missing values**

```python
In [3]: outer_join.dropna()
```

```
Out[3]:
key  value_x  value_y
1  B  -0.282863  1.212112
3  D  -1.135632 -0.173215
4  D  -1.135632  0.119209
```

**Forward fill from previous rows**

```python
In [4]: outer_join.fillna(method="ffill")
```

```
Out[4]:
key  value_x  value_y
0  A   0.469112     NaN
1  B  -0.282863  1.212112
2  C  -1.509059  1.212112
3  D  -1.135632 -0.173215
4  D  -1.135632  0.119209
5  E  -1.135632 -1.044236
```

**Replace missing values with a specified value**

Using the mean:

```python
In [1]: outer_join["value_x"].fillna(outer_join["value_x"].mean())
```

```
Out[1]:
0   0.469112
1  -0.282863
2  -1.509059
3  -1.135632
4  -1.135632
5  -0.718815
Name: value_x, dtype: float64
```
**GroupBy**

**Aggregation**

SAS’s PROC SUMMARY can be used to group by one or more key variables and compute aggregations on numeric columns.

```plaintext
proc summary data=tips nway;
   class sex smoker;
   var total_bill tip;
   output out=tips_summed sum=;
run;
```

pandas provides a flexible groupby mechanism that allows similar aggregations. See the groupby documentation for more details and examples.

```
In [1]: tips_summed = tips.groupby(["sex", "smoker"])[["total_bill", "tip"]].sum()
```

```
In [2]: tips_summed
```

```
Out[2]:
      total_bill  tip
    sex smoker       
Female No  869.68 149.77
           Yes  527.27  96.74
Male   No  1725.75 302.00
           Yes 1217.07 183.07
```

**Transformation**

In SAS, if the group aggregations need to be used with the original frame, it must be merged back together. For example, to subtract the mean for each observation by smoker group.

```plaintext
proc summary data=tips missing nway;
   class smoker;
   var total_bill;
   output out=smoker_means mean(total_bill)=group_bill;
run;

proc sort data=tips;
   by smoker;
run;

data tips;
   merge tips(in=a) smoker_means(in=b);
   by smoker;
   adj_total_bill = total_bill - group_bill;
   if a and b;
run;
```

pandas provides a Transformation mechanism that allows these type of operations to be succinctly expressed in one operation.
In [1]: gb = tips.groupby("smoker")["total_bill"]

In [2]: tips["adj_total_bill"] = tips["total_bill"] - gb.transform("mean")

In [3]: tips
Out[3]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>adj_total_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>1.07</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-17.686344</td>
</tr>
<tr>
<td>92</td>
<td>3.75</td>
<td>Female</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>2</td>
<td>-15.006344</td>
</tr>
<tr>
<td>111</td>
<td>5.25</td>
<td>Female</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-11.938278</td>
</tr>
<tr>
<td>145</td>
<td>6.35</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
<td>-10.838278</td>
</tr>
<tr>
<td>135</td>
<td>6.51</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
<td>-10.678278</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>182</td>
<td>43.35</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>24.593656</td>
</tr>
<tr>
<td>156</td>
<td>46.17</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>6</td>
<td>28.981722</td>
</tr>
<tr>
<td>59</td>
<td>46.27</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
<td>29.081722</td>
</tr>
<tr>
<td>212</td>
<td>46.33</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
<td>29.141722</td>
</tr>
<tr>
<td>170</td>
<td>48.81</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
<td>30.053656</td>
</tr>
<tr>
<td>244 rows x 8 columns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By group processing

In addition to aggregation, pandas groupby can be used to replicate most other by group processing from SAS. For example, this DATA step reads the data by sex/smoker group and filters to the first entry for each.

```plaintext
proc sort data=tips;
  by sex smoker;
run;

data tips_first;
  set tips;
  by sex smoker;
  if FIRST.sex or FIRST.smoker then output;
run;
```

In pandas this would be written as:

In [4]: tips.groupby(["sex", "smoker"]).first()

Out[4]:

<table>
<thead>
<tr>
<th>sex</th>
<th>smoker</th>
<th>total_bill</th>
<th>tip</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>adj_total_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Female</td>
<td>No</td>
<td>5.25</td>
<td>1.00</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-11.938278</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>1.07</td>
<td>1.00</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-17.686344</td>
</tr>
<tr>
<td>Male</td>
<td>No</td>
<td>5.51</td>
<td>2.00</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
<td>-11.678278</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>5.25</td>
<td>5.15</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>-13.506344</td>
</tr>
</tbody>
</table>
Other considerations

Disk vs memory

pandas operates exclusively in memory, where a SAS data set exists on disk. This means that the size of data able to be loaded in pandas is limited by your machine’s memory, but also that the operations on that data may be faster.

If out of core processing is needed, one possibility is the dask.dataframe library (currently in development) which provides a subset of pandas functionality for an on-disk DataFrame.

Data interop

pandas provides a read_sas() method that can read SAS data saved in the XPORT or SAS7BDAT binary format.

```python
libname xportout xport 'transport-file.xpt';
data xportout.tips;
   set tips(rename=(total_bill=tbill));
   * xport variable names limited to 6 characters;
run;

df = pd.read_sas("transport-file.xpt")
df = pd.read_sas("binary-file.sas7bdat")
```

You can also specify the file format directly. By default, pandas will try to infer the file format based on its extension.

```python
df = pd.read_sas("transport-file.xpt", format="xport")
df = pd.read_sas("binary-file.sas7bdat", format="sas7bdat")
```

XPORT is a relatively limited format and the parsing of it is not as optimized as some of the other pandas readers. An alternative way to interop data between SAS and pandas is to serialize to csv.

# version 0.17, 10M rows

```python
In [8]: %time df = pd.read_sas('big.xpt')
Wall time: 14.6 s

In [9]: %time df = pd.read_csv('big.csv')
Wall time: 4.86 s
```

Comparison with Stata

For potential users coming from Stata this page is meant to demonstrate how different Stata operations would be performed in pandas.

If you’re new to pandas, you might want to first read through 10 Minutes to pandas to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```python
In [1]: import pandas as pd
In [2]: import numpy as np
```
Data structures

General terminology translation

<table>
<thead>
<tr>
<th>pandas</th>
<th>Stata</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame</td>
<td>data set</td>
</tr>
<tr>
<td>column</td>
<td>variable</td>
</tr>
<tr>
<td>row</td>
<td>observation</td>
</tr>
<tr>
<td>groupby</td>
<td>bysort</td>
</tr>
<tr>
<td>NaN</td>
<td>.</td>
</tr>
</tbody>
</table>

DataFrame

A DataFrame in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

Series

A Series is the data structure that represents one column of a DataFrame. Stata doesn’t have a separate data structure for a single column, but in general, working with a Series is analogous to referencing a column of a data set in Stata.

Index

Every DataFrame and Series has an Index – labels on the rows of the data. Stata does not have an exactly analogous concept. In Stata, a data set’s rows are essentially unlabeled, other than an implicit integer index that can be accessed with _n.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the Index and just treat the DataFrame as a collection of columns. Please see the indexing documentation for much more on how to use an Index effectively.

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes “stick”, you’ll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

Note: You will see an inplace=True keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```
Data input / output

Constructing a DataFrame from values

A Stata data set can be built from specified values by placing the data after an `input` statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```python
In [3]: df = pd.DataFrame({"x": [1, 3, 5], "y": [2, 4, 6]})
```

```python
In [4]: df
Out[4]:
x y
0 1 2
1 3 4
2 5 6
```

Reading external data

Like Stata, pandas provides utilities for reading in data from many formats. The tips data set, found within the pandas tests (csv) will be used in many of the following examples.

Stata provides `import delimited` to read csv data into a data set in memory. If the `tips.csv` file is in the current working directory, we can import it as follows.

```python
import delimited tips.csv
```

The pandas method is `read_csv()`, which works similarly. Additionally, it will automatically download the data set if presented with a url.

```python
In [5]: url = (
    ...:   "https://raw.github.com/pandas-dev"
    ...:   "https://raw.github.com/pandas-dev"
    ...:   "https://raw.github.com/pandas-dev"
    ...: )
    ...

In [6]: tips = pd.read_csv(url)
```

```python
In [7]: tips
Out[7]:
total_bill  tip  sex  smoker  day  time  size
        (continues on next page)
Like import delimited, read_csv() can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```python
tips = pd.read_csv("tips.csv", sep="\t", header=None)
```

pandas can also read Stata data sets in .dta format with the read_stata() function.

```python
df = pd.read_stata("data.dta")
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a pd.read_* function. See the IO documentation for more details.

### Limiting output

By default, pandas will truncate output of large DataFrames to show the first and last rows. This can be overridden by changing the pandas options, or using DataFrame.head() or DataFrame.tail().

```python
In [8]: tips.head(5)
Out[8]:
   total_bill  tip  sex   smoker day  time  size
0       16.99  1.01 Female   No  Sun  Dinner  2
1       10.34  1.66   Male   No  Sun  Dinner  3
2       21.01  3.50   Male   No  Sun  Dinner  3
3       23.68  3.31   Male   No  Sun  Dinner  2
4       24.59  3.61 Female   No  Sun  Dinner  4
```

The equivalent in Stata would be:

```
list in 1/5
```
Exporting data

The inverse of `import delimited` in Stata is `export delimited`

```python
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is `DataFrame.to_csv()`.

```python
tips.to_csv("tips2.csv")
```

Pandas can also export to Stata file format with the `DataFrame.to_stata()` method.

```python
tips.to_stata("tips2.dta")
```

Data operations

Operations on columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```stata
replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill
```

Pandas provides vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`.

```python
In [9]: tips["total_bill"] = tips["total_bill"] - 2

In [10]: tips["new_bill"] = tips["total_bill"] / 2

In [11]: tips
Out[11]:
     total_bill  tip  sex  smoker day  time  size  new_bill
0     14.99    1.01  Female  No  Sun  Dinner   2   7.495
1      8.34    1.66    Male  No  Sun  Dinner   3   4.170
2     19.01    3.50    Male  No  Sun  Dinner   3   9.505
3     21.68    3.31    Male  No  Sun  Dinner   2  10.840
4     22.59    3.61  Female  No  Sun  Dinner   4  11.295
...     ...     ...     ...  ...  ...  ...     ...  ...
239    27.03    5.92    Male  No  Sat  Dinner   3  13.515
240    25.18    2.00  Female   Yes  Sat  Dinner   2  12.590
241    20.67    2.00    Male  Yes  Sat  Dinner   2  10.335
242    15.82    1.75    Male  No  Sat  Dinner   2   7.910
243    16.78    3.00  Female  No  Thur  Dinner   2   8.390

[244 rows x 8 columns]

In [12]: tips = tips.drop("new_bill", axis=1)
Filtering

Filtering in Stata is done with an if clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```
In [13]: tips[tips["total_bill"] > 10]
Out [13]:
  total_bill  tip  sex  smoker  day  time  size
  0  14.99  1.01 Female  No  Sun  Dinner  2
  2  19.01  3.50  Male  No  Sun  Dinner  3
  3  21.68  3.31  Male  No  Sun  Dinner  2
  4  22.59  3.61 Female  No  Sun  Dinner  4
  5  23.29  4.71  Male  No  Sun  Dinner  4
  ...
  239  27.03  5.92  Male  No  Sat  Dinner  3
  240  25.18  2.00 Female  Yes  Sat  Dinner  2
  241  20.67  2.00  Male  Yes  Sat  Dinner  2
  242  15.82  1.75  Male  No  Sat  Dinner  2
  243  16.78  3.00 Female  No  Thur  Dinner  2

[204 rows x 7 columns]
```

The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

```
In [14]: is_dinner = tips["time"] == "Dinner"
In [15]: is_dinner
Out [15]:
 0   True
 1   True
 2   True
 3   True
 4   True
   ...  ...  ...  ...  ...  ...  ...
239 True
240 True
241 True
242 True
243 True
Name: time, Length: 244, dtype: bool

In [16]: is_dinner.value_counts()
Out [16]:
True  176
False  68
Name: time, dtype: int64

In [17]: tips[is_dinner]
Out [17]:
```

(continues on next page)
If/then logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [18]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")
```

```
In [19]: tips
Out[19]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
<tr>
<td>8.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>low</td>
</tr>
<tr>
<td>19.01</td>
<td>3.50</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>high</td>
</tr>
<tr>
<td>21.68</td>
<td>3.31</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
<tr>
<td>22.59</td>
<td>3.61</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
<td>high</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>27.03</td>
<td>5.92</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
<td>high</td>
</tr>
<tr>
<td>25.18</td>
<td>2.00</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
<tr>
<td>20.67</td>
<td>2.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
<tr>
<td>15.82</td>
<td>1.75</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
<tr>
<td>16.78</td>
<td>3.00</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Dinner</td>
<td>2</td>
<td>high</td>
</tr>
</tbody>
</table>

[244 rows x 8 columns]

Date functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")
generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)
```

```
list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the timeseries documentation for more details.

```
In [20]: tips["date1"] = pd.Timestamp("2013-01-15")
In [21]: tips["date2"] = pd.Timestamp("2015-02-15")
In [22]: tips["date1_year"] = tips["date1"]).dt.year
In [23]: tips["date2_month"] = tips["date2"]).dt.month
In [24]: tips["date1_next"] = tips["date1"] + pd.offsets.MonthBegin()
In [25]: tips["months_between"] = tips["date2"]).dt.to_period("M") - tips["date1").dt.to_period("M")
```

```
Out[26]:
      date1    date2  date1_year  date2_month  date1_next  months_between
...        ...       ...        ...          ...        ...
```

(continues on next page)
Selection of columns

Stata provides keywords to select, drop, and rename columns.

```
keep  sex  total_bill  tip
drop  sex
rename total_bill  total_bill_2
```

The same operations are expressed in pandas below.

**Keep certain columns**

```
In [27]: tips["sex", "total_bill", "tip"]
Out[27]:
   sex   total_bill  tip
0  Female   14.99    1.01
1    Male    8.34    1.66
2    Male   19.01    3.50
3    Male   21.68    3.31
4  Female   22.59    3.61
   ...     ...     ...
239  Male   27.03    5.92
240 Female   25.18    2.00
241    Male   20.67    2.00
242    Male   15.82    1.75
243 Female   16.78    3.00
```

[244 rows x 3 columns]

**Drop a column**

```
In [28]: tips.drop("sex", axis=1)
Out[28]:
   total_bill  tip  smoker  day  time  size
0    14.99    1.01  No   Sun  Dinner  2
1     8.34    1.66  No   Sun  Dinner  3
2    19.01    3.50  No   Sun  Dinner  3
3    21.68    3.31  No   Sun  Dinner  2
4    22.59    3.61  No   Sun  Dinner  4
   ...     ...     ...     ...     ...     ...
239   27.03    5.92  Yes  Sat  Dinner  3
240   25.18    2.00   Yes  Sat  Dinner  2
```

(continues on next page)
Rename a column

In [29]: tips.rename(columns={"total_bill": "total_bill_2"})

Out[29]:
    total_bill_2  tip  sex  smoker  day    time  size
0      14.99   1.01 Female  No  Sun  Dinner  2
1       8.34   1.66  Male   No  Sun  Dinner  3
2      19.01   3.50  Male   No  Sun  Dinner  3
3      21.68   3.31  Male   No  Sun  Dinner  2
4      22.59   3.61 Female  No  Sun  Dinner  4
...     ...     ...     ...     ...     ...    ...
239    27.03   5.92  Male   No  Sat  Dinner  3
240    25.18   2.00 Female  Yes  Sun  Dinner  2
241    20.67   2.00  Male   Yes  Sat  Dinner  2
242    15.82   1.75  Male   No  Sat  Dinner  2
243    16.78   3.00 Female  No  Thur  Dinner  2
[244 rows x 7 columns]

Sorting by values

Sorting in Stata is accomplished via sort

```
sort sex total_bill
```

pandas has a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

In [30]: tips = tips.sort_values(["sex", "total_bill"])

In [31]: tips

Out[31]:
    total_bill  tip  sex  smoker  day    time  size
67      1.07   1.00 Female  Yes  Sat  Dinner  1
92      3.75   1.00 Female  Yes  Fri  Dinner  2
111     5.25   1.00 Female  No  Sat  Dinner  1
145     6.35   1.50 Female  No  Thur  Lunch  2
135     6.51   1.25 Female  No  Thur  Lunch  2
...     ...     ...     ...     ...     ...    ...
182     43.35   3.50  Male   Yes  Sun  Dinner  3
156     46.17   5.00  Male   No  Sun  Dinner  6
59      46.27   6.73  Male   No  Sat  Dinner  4
212     46.33   9.00  Male   No  Sat  Dinner  4
170     48.81  10.00  Male   Yes  Sat  Dinner  3
[244 rows x 7 columns]
String processing

Finding length of string

Stata determines the length of a character string with the \texttt{strlen()} and \texttt{ustrlen()} functions for ASCII and Unicode strings, respectively.

\begin{verbatim}
generate strlen_time = strlen(time)
generate ulstrlen_time = ustrlen(time)
\end{verbatim}

You can find the length of a character string with \texttt{Series.str.len()}. In Python 3, all strings are Unicode strings. \texttt{len} includes trailing blanks. Use \texttt{len} and \texttt{rstrip} to exclude trailing blanks.

\begin{verbatim}
In [32]: tips["time"].str.len()
Out[32]:
   67    6
   92    6
  111    6
  145    5
  135    5
     ..
  182    6
  156    6
   59    6
  212    6
  170    6
Name: time, Length: 244, dtype: int64

In [33]: tips["time"].str.rstrip().str.len()
Out[33]:
   67    6
   92    6
  111    6
  145    5
  135    5
     ..
  182    6
  156    6
   59    6
  212    6
  170    6
Name: time, Length: 244, dtype: int64
\end{verbatim}
Finding position of substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```plaintext
generate str_position = strpos(sex, "ale")
```

You can find the position of a character in a column of strings with the `Series.str.find()` method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns -1. Keep in mind that Python indexes are zero-based.

```plaintext
In [34]: tips["sex"].str.find("ale")
Out[34]:
67  3
92  3
111 3
145 3
135 3
   ...
182 1
156 1
59  1
212 1
170 1
Name: sex, Length: 244, dtype: int64
```

Extracting substring by position

Stata extracts a substring from a string based on its position with the `substr()` function.

```plaintext
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```plaintext
In [35]: tips["sex"][0:1]
Out[35]:
67   F
92   F
111  F
145  F
135  F
   ...
182  M
156  M
59   M
212  M
170  M
Name: sex, Length: 244, dtype: object
```
Extracting nth word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```plaintext
clear
input str20 string
 "John Smith"
 "Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

```plaintext
In [36]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})
In [37]: firstlast["First_Name"] = firstlast["String"].str.split(" ", expand=True)[0]
In [38]: firstlast["Last_Name"] = firstlast["String"].str.rsplit(" ", expand=True)[1]
In [39]: firstlast
Out[39]:
          String    First_Name    Last_Name
0   John Smith    John          Smith
1      Jane Cook      Jane        Cook
```

Changing case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```plaintext
clear
input str20 string
 "John Smith"
 "Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent pandas methods are `Series.str.upper()`, `Series.str.lower()`, and `Series.str.title()`.

```plaintext
In [40]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})
In [41]: firstlast["upper"] = firstlast["string"].str.upper()
In [42]: firstlast["lower"] = firstlast["string"].str.lower()
```
In [43]: `firstlast["title"] = firstlast["string"].str.title()

In [44]: `firstlast

Out[44]:
   string       upper       lower       title
0  John Smith    JOHN SMITH  john smith  John Smith
1     Jane Cook    JANE COOK  jane cook Jane Cook

Merging

The following tables will be used in the merge examples:

In [45]: `df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [46]: `df1

Out[46]:
   key   value
0   A   0.469112
1   B  -0.282863
2   C  -1.509059
3   D  -1.135632

In [47]: `df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

In [48]: `df2

Out[48]:
   key   value
0   B   1.212112
1   D  -0.173215
2   D   0.119209
3   E  -1.044236

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both DataFrames already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the _merge variable.
* Now create df1 in memory

clear
input str1 key
A
B
C
D
end
generate value = rnormal()
preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta

Pandas DataFrames have a `merge()` method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

In [49]: inner_join = df1.merge(df2, on="key", how="inner")

In [50]: inner_join
Out[50]:
key     value_x     value_y
0   B   -0.282863  1.212112
1   D  -1.135632  -0.173215
2   D  -1.135632   0.119209

In [51]: left_join = df1.merge(df2, on="key", how="left")

In [52]: left_join
Out[52]:
key     value_x     value_y
0   A   0.469112    NaN
1   B  -0.282863  1.212112
2   C  -1.509059    NaN
3   D  -1.135632  -0.173215
4   D  -1.135632   0.119209
**Missing data**

Both pandas and Stata have a representation for missing data.

pandas represents missing data with the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.
One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```plaintext
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

In pandas, `Series.isna()` and `Series.notna()` can be used to filter the rows.

```plaintext
In [60]: outer_join[outer_join["value_x"].isna()]
Out[60]:
      key  value_x  value_y     
   5    E      NaN  -1.044236

In [61]: outer_join[outer_join["value_x"].notna()]
Out[61]:
      key  value_x  value_y     
   0    A  0.469112  NaN     
   1    B  1.282863  1.212112 
   2    C -1.509059  NaN     
   3    D -1.135632  0.173215 
   4    D -1.135632  0.119209 
   5    E -1.135632 -1.044236 
```

pandas provides a variety of methods to work with missing data. Here are some examples:

**Drop rows with missing values**

```plaintext
In [62]: outer_join.dropna()
Out[62]:
      key  value_x  value_y     
   1    B  1.282863  1.212112 
   3    D -1.135632  0.173215 
   4    D -1.135632  0.119209 
```

**Forward fill from previous rows**

```plaintext
In [63]: outer_join.fillna(method="ffill")
Out[63]:
      key  value_x  value_y     
   0    A  0.469112  NaN     
   1    B  1.282863  1.212112 
   2    C -1.509059  1.212112 
   3    D -1.135632  0.173215 
   4    D -1.135632  0.119209 
   5    E -1.135632 -1.044236 
```
Replace missing values with a specified value

Using the mean:

```python
In [64]: outer_join["value_x"].fillna(outer_join["value_x"].mean())
Out[64]:
     0  0.469112
     1 -0.282863
     2 -1.509059
     3 -1.135632
     4 -1.135632
     5 -0.718815
Name: value_x, dtype: float64
```

**GroupBy**

**Aggregation**

Stata's `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the `groupby documentation` for more details and examples.

```python
In [65]: tips_summed = tips.groupby(["sex", "smoker"])[["total_bill", "tip"]].sum()
In [66]: tips_summed
Out[66]:
            total_bill  tip
   sex smoker       869.68  149.77
     Female No    527.27   96.74
     Yes         1725.75  302.00
   Male No       1217.07  183.07
     Yes         1217.07  183.07
```

**Transformation**

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker:
     egen group_bill = mean(total_bill)
     generate adj_total_bill = total_bill - group_bill
```

pandas provides a `Transformation` mechanism that allows these type of operations to be succinctly expressed in one operation.

```python
In [67]: gb = tips.groupby("smoker")["total_bill"]
In [68]: tips["adj_total_bill"] = tips["total_bill"] - gb.transform("mean")
```
In [69]: tips
Out[69]:

<table>
<thead>
<tr>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>adj_total_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>1.07</td>
<td>1.00</td>
<td>Female</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
</tr>
<tr>
<td>92</td>
<td>3.75</td>
<td>1.00</td>
<td>Female</td>
<td>Yes</td>
<td>Fri</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>111</td>
<td>5.25</td>
<td>1.00</td>
<td>Female</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
</tr>
<tr>
<td>145</td>
<td>6.35</td>
<td>1.50</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
</tr>
<tr>
<td>135</td>
<td>6.51</td>
<td>1.25</td>
<td>Female</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>182</td>
<td>43.35</td>
<td>3.50</td>
<td>Male</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>156</td>
<td>46.17</td>
<td>5.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td>46.27</td>
<td>6.73</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>212</td>
<td>46.33</td>
<td>9.00</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>4</td>
</tr>
<tr>
<td>170</td>
<td>48.81</td>
<td>10.00</td>
<td>Male</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
</tbody>
</table>

[244 rows x 8 columns]

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [70]: tips.groupby(["sex", "smoker"]).first()
Out[70]:

<table>
<thead>
<tr>
<th>sex</th>
<th>smoker</th>
<th>total_bill</th>
<th>tip</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>adj_total_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Female</td>
<td>No</td>
<td>5.25</td>
<td>1.00</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-11.938278</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td>1.07</td>
<td>1.00</td>
<td>Sat</td>
<td>Dinner</td>
<td>1</td>
<td>-17.686344</td>
</tr>
<tr>
<td>Male</td>
<td>No</td>
<td>5.51</td>
<td>2.00</td>
<td>Thur</td>
<td>Lunch</td>
<td>2</td>
<td>-11.678278</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>5.25</td>
<td>5.15</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>-13.506344</td>
</tr>
</tbody>
</table>
```

Other considerations

Disk vs memory

pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine’s memory. If out of core processing is needed, one possibility is the `dask.dataframe` library, which provides a subset of pandas functionality for an on-disk DataFrame.
1.4.5 Community tutorials

This is a guide to many pandas tutorials by the community, geared mainly for new users.

pandas cookbook by Julia Evans

The goal of this 2015 cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails. For the table of contents, see the pandas-cookbook GitHub repository.

pandas workshop by Stefanie Molin

An introductory workshop by Stefanie Molin designed to quickly get you up to speed with pandas using real-world datasets. It covers getting started with pandas, data wrangling, and data visualization (with some exposure to matplotlib and seaborn). The pandas-workshop GitHub repository features detailed environment setup instructions (including a Binder environment), slides and notebooks for following along, and exercises to practice the concepts. There is also a lab with new exercises on a dataset not covered in the workshop for additional practice.

Learn pandas by Hernan Rojas

A set of lesson for new pandas users: https://bitbucket.org/hrojas/learn-pandas

Practical data analysis with Python

This guide is an introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as munging data, aggregating data, visualizing data and time series.

Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

Modern pandas

Tutorial series written in 2016 by Tom Augspurger. The source may be found in the GitHub repository TomAugspurger/effective-pandas.

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization
- Timeseries
Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

Video tutorials

- Pandas: .head() to .tail() (2016) (1:26) GitHub repo
- Data analysis in Python with pandas (2016-2018) GitHub repo and Jupyter Notebook
- Best practices with pandas (2018) GitHub repo and Jupyter Notebook

Various tutorials

- Wes McKinney’s (pandas BDFL) blog
- Statistical analysis made easy in Python with SciPy and pandas DataFrames, by Randal Olson
- Statistical Data Analysis in Python, tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- Financial analysis in Python, by Thomas Wiecki
- Intro to pandas data structures, by Greg Reda
- Pandas and Python: Top 10, by Manish Amde
- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples
The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as “working with missing data”), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

For a high level summary of the pandas fundamentals, see *Intro to data structures* and *Essential basic functionality*.

Further information on any specific method can be obtained in the *API reference*.

### 2.1 10 minutes to pandas

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*.

Customarily, we import as follows:

```python
In [1]: import numpy as np
In [2]: import pandas as pd
```

#### 2.1.1 Object creation

See the *Intro to data structures section*.

Creating a *Series* by passing a list of values, letting pandas create a default integer index:

```python
In [3]: s = pd.Series([1, 3, 5, np.nan, 6, 8])
In [4]: s
Out[4]:
0  1.0
1  3.0
2  5.0
3  NaN
4  6.0
5  8.0
dtype: float64
```

Creating a *DataFrame* by passing a NumPy array, with a datetime index and labeled columns:
In [5]: dates = pd.date_range("20130101", periods=6)

In [6]: dates

Out[6]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
             dtype='datetime64[ns]', freq='D')

In [7]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

In [8]: df

Out[8]:
     A      B      C      D
2013-01-01 -0.421613 0.429509 -0.325377 0.937242
2013-01-02 -0.004552 -0.995324 0.365204 0.930581
2013-01-03  0.595855 1.133877  0.109665 0.130624
2013-01-04 -0.178563 -1.369634 -0.562609 0.193083
2013-01-05 -0.052989  1.368210 -1.233963 0.563247
2013-01-06  1.062178 -0.798260 -0.237608 -0.349017

Creating a DataFrame by passing a dictionary of objects that can be converted into a series-like structure:

In [9]: df2 = pd.DataFrame(
    ...:     {
    ...:         "A": 1.0,
    ...:         "B": pd.Timestamp("20130102"),
    ...:         "C": pd.Series(1, index=list(range(4)), dtype="float32"),
    ...:         "D": np.array([3] * 4, dtype="int32"),
    ...:         "E": pd.Categorical(["test", "train", "test", "train"]),
    ...:         "F": "foo",
    ...:     })

In [10]: df2

Out[10]:
     A      B      C      D      E      F
0  1.0  2013-01-02  1.0   3.0  test  foo
1  1.0  2013-01-02  1.0   3.0  train foo
2  1.0  2013-01-02  1.0   3.0  test  foo
3  1.0  2013-01-02  1.0   3.0  train foo

The columns of the resulting DataFrame have different dtypes:

In [11]: df2.dtypes

Out[11]:
a       float64
b  datetime64[ns]
c       float32
d        int32
e     category
f         object
dtype: object
If you’re using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here’s a subset of the attributes that will be completed:

```latex
\begin{Verbatim}
In [12]: df2.<TAB>  # noqa: E225, E999
    df2.A          df2.bool
    df2.abs        df2.boxplot
    df2.add        df2.C
    df2.add_prefix df2.clip
    df2.add_suffix df2.columns
    df2.align      df2.copy
    df2.all        df2.count
    df2.any        df2.combine
    df2.append     df2.D
    df2.apply      df2.describe
    df2.applymap   df2.diff
    df2.B          df2.duplicated
\end{Verbatim}
```

As you can see, the columns A, B, C, and D are automatically tab completed. E and F are there as well; the rest of the attributes have been truncated for brevity.

## 2.1.2 Viewing data

See the Basics section.

Here is how to view the top and bottom rows of the frame:

```latex
\begin{Verbatim}
In [13]: df.head()
Out[13]:
      A       B       C       D
2013-01-01 -0.421613 0.429509 -0.325377 0.937242
2013-01-02 -0.004552 -0.995324 0.365204 0.930581
2013-01-03  0.595855  1.133877  0.109665  0.130624
2013-01-04 -0.178563 -1.369634 -0.562609  0.193083
2013-01-05 -0.052989  1.368210 -1.233963  0.563247

In [14]: df.tail(3)
Out[14]:
      A       B       C       D
2013-01-04 -0.178563 -1.369634 -0.562609  0.193083
2013-01-05 -0.052989  1.368210 -1.233963  0.563247
2013-01-06  1.062178 -0.798260 -0.237608 -0.349017
\end{Verbatim}
```

Display the index, columns:

```latex
\begin{Verbatim}
In [15]: df.index
Out[15]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
'2013-01-05', '2013-01-06'],
dtype='datetime64[ns]', freq='D')

In [16]: df.columns
Out[16]: Index(['A', 'B', 'C', 'D'], dtype='object')
\end{Verbatim}
```

DataFrame.to_numpy() gives a NumPy representation of the underlying data. Note that this can be an expensive
operation when your DataFrame has columns with different data types, which comes down to a fundamental difference between pandas and NumPy: **NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column.** When you call DataFrame.to_numpy(), pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. This may end up being object, which requires casting every value to a Python object.

For df, our DataFrame of all floating-point values, DataFrame.to_numpy() is fast and doesn’t require copying data:

```python
In [17]: df.to_numpy()
```
```
Out[17]:
array([[-0.42161269,  0.42950902, -0.32537683,  0.93724182],
       [-0.00455221, -0.99532411,  0.36520398,  0.93058093],
       [ 0.59585469,  1.1338767 ,  0.10966549,  0.1306242 ],
       [-0.17856322, -1.36963422, -0.56260921,  0.19308313],
       [-0.05298949,  1.36821038, -1.23396251,  0.56324681],
       [ 1.06217759, -0.79826025, -0.23760797, -0.34901652]])
```

For df2, the DataFrame with multiple dtypes, DataFrame.to_numpy() is relatively expensive:

```python
In [18]: df2.to_numpy()
```
```
Out[18]:
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

**Note:** DataFrame.to_numpy() does not include the index or column labels in the output.

describe() shows a quick statistic summary of your data:

```python
In [19]: df.describe()
```
```
Out[19]:
                       A         B         C         D
count        6.000000  6.000000  6.000000  6.000000
mean       0.166719 -0.038604 -0.314115  0.400960
std        0.552985  1.169359  0.557658  0.504617
min       -0.421613 -1.369634 -1.233963 -0.349017
25%        -0.147170 -0.946058 -0.503301  0.146239
50%        -0.028771 -0.184376 -0.281492  0.378165
75%        0.445753  0.957785  0.022847  0.838747
max        1.062178  1.368210  0.365204  0.937242
```

Transposing your data:

```python
In [20]: df.T
```
```
Out[20]:
A     -0.421613   -0.004552     0.595855  -0.178563    -0.052989     1.062178
B      0.429509   -0.995324     1.133877  -1.369634    -0.562609    -0.798260
C     -0.325377     0.365284    0.109665  -0.562609    -1.233963    -0.237608
D      0.937242     0.930581     0.130624     0.193083     0.563247    -0.349017
```

Sorting by an axis:
In [21]: df.sort_index(axis=1, ascending=False)

Out[21]:
          D     C     B    A
2013-01-01  0.937242 -0.325377 0.429509 -0.421613
2013-01-02  0.930581  0.365204 -0.995324 -0.004552
2013-01-03  0.130624  0.109665  1.133877  0.595855
2013-01-04  0.193083 -0.562609 -1.369634 -0.178563
2013-01-05  0.563247 -1.233963  1.368210 -0.052989
2013-01-06 -0.349017 -0.237608 -0.798260  1.062178

Sorting by values:

In [22]: df.sort_values(by="B")

Out[22]:
          A     B     C     D
2013-01-04 -0.178563 -1.369634 -0.562609  0.193083
2013-01-02 -0.004552 -0.995324  0.365204  0.930581
2013-01-06  1.062178 -0.798260 -0.237608 -0.349017
2013-01-01 -0.421613  0.429509 -0.325377  0.937242
2013-01-03  0.595855  1.133877  0.109665  0.130624
2013-01-05 -0.052989  1.368210 -1.233963  0.563247

2.1.3 Selection

Note: While standard Python / NumPy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, .at, .iat, .loc and .iloc.

See the indexing documentation Indexing and Selecting Data and MultiIndex / Advanced Indexing.

Getting

Selecting a single column, which yields a Series, equivalent to df.A:

In [23]: df["A"]

Out[23]:
2013-01-01    -0.421613
2013-01-02    -0.004552
2013-01-03     0.595855
2013-01-04    -0.178563
2013-01-05    -0.052989
2013-01-06     1.062178
Freq: D, Name: A, dtype: float64

Selecting via [], which slices the rows:

In [24]: df[0:3]

Out[24]:
          A     B     C     D
2013-01-01    -0.421613  0.429509 -0.325377  0.937242
          (continues on next page)
2013-01-02 -0.004552 -0.995324 0.365204 0.930581
2013-01-03 0.595855 1.133877 0.109665 0.130624

In [25]: df["20130102":"20130104"]
Out[25]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>0.365204</td>
<td>0.930581</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>0.130624</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
<td>-0.562609</td>
<td>0.193083</td>
</tr>
</tbody>
</table>

Selection by label

See more in Selection by Label.

For getting a cross section using a label:

In [26]: df.loc[dates[0]]
Out[26]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>-0.421613</td>
<td>0.429509</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.052989</td>
<td>1.368210</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>1.062178</td>
<td>-0.798260</td>
</tr>
</tbody>
</table>

Name: 2013-01-01 00:00:00, dtype: float64

Selecting on a multi-axis by label:

In [27]: df.loc[:,["A","B"]]
Out[27]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>-0.421613</td>
<td>0.429509</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.052989</td>
<td>1.368210</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>1.062178</td>
<td>-0.798260</td>
</tr>
</tbody>
</table>

Showing label slicing, both endpoints are included:

In [28]: df.loc["20130102":"20130104",["A","B"]]
Out[28]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
</tr>
</tbody>
</table>

Reduction in the dimensions of the returned object:

In [29]: df.loc["20130102",["A","B"]]
Out[29]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
</tr>
<tr>
<td>Name: 2013-01-02 00:00:00, dtype: float64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For getting a scalar value:

```python
In [30]: df.loc[dates[0], "A"]
Out[30]: -0.42161268764456333
```

For getting fast access to a scalar (equivalent to the prior method):

```python
In [31]: df.at[dates[0], "A"]
Out[31]: -0.42161268764456333
```

### Selection by position

See more in *Selection by Position*.

Select via the position of the passed integers:

```python
In [32]: df.iloc[3]
Out[32]:
A   -0.178563
B   -1.369634
C   -0.562609
D    0.193083
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to NumPy/Python:

```python
In [33]: df.iloc[3:5, 0:2]
Out[33]:
       A      B
2013-01-04 -0.178563 -1.369634
2013-01-05 -0.052989  1.368210
```

By lists of integer position locations, similar to the NumPy/Python style:

```python
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
       A      C
2013-01-02 -0.004552  0.365204
2013-01-03  0.595855  0.109665
2013-01-05 -0.052989 -1.233963
```

For slicing rows explicitly:

```python
In [35]: df.iloc[1:3, :]
Out[35]:
       A      B      C      D
2013-01-02 -0.004552 -0.995324  0.365204  0.930581
2013-01-03  0.595855  1.133877  0.109665  0.130624
```

For slicing columns explicitly:

```python
In [36]: df.iloc[:, 1:3]
Out[36]:
       B      C
```

(continues on next page)
For getting a value explicitly:

```python
In [37]: df.iloc[1, 1]
Out[37]: -0.9953241139603133
```

For getting fast access to a scalar (equivalent to the prior method):

```python
In [38]: df.iat[1, 1]
Out[38]: -0.9953241139603133
```

### Boolean indexing

Using a single column’s values to select data:

```python
In [39]: df[df["A"] > 0]
Out[39]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>0.130624</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>1.062178</td>
<td>-0.798260</td>
<td>-0.237608</td>
<td>-0.349017</td>
</tr>
</tbody>
</table>
```

Selecting values from a DataFrame where a boolean condition is met:

```python
In [40]: df[df > 0]
Out[40]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>NaN</td>
<td>0.429509</td>
<td>NaN</td>
<td>0.937242</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>0.365204</td>
<td>0.930581</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>0.130624</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>0.193083</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>NaN</td>
<td>1.368210</td>
<td>NaN</td>
<td>0.563247</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>1.062178</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

Using the `isin()` method for filtering:

```python
In [41]: df2 = df.copy()
In [42]: df2["E"] = ["one", "one", "two", "three", "four", "three"]
In [43]: df2
Out[43]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>-0.421613</td>
<td>0.429509</td>
<td>-0.325377</td>
<td>0.937242</td>
<td>one</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>0.365204</td>
<td>0.930581</td>
<td>one</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>0.130624</td>
<td>two</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
<td>-0.562609</td>
<td>0.193083</td>
<td>three</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Setting a new column automatically aligns the data by the indexes:

```
In [45]: s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20130102", periods=6))
In [46]: s1
```

```
Out[46]:
2013-01-02 1
2013-01-03 2
2013-01-04 3
2013-01-05 4
2013-01-06 5
2013-01-07 6
Freq: D, dtype: int64
```

```
In [47]: df["F"] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], "A"] = 0
```

Setting values by position:

```
In [49]: df.iat[0, 1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, "D"] = np.array([5] * len(df))
```

The result of the prior setting operations:

```
In [51]: df
Out[51]:
    A        B        C        D       E
2013-01-01 0.000000 0.000000 -0.325377 5.000000 NaN
2013-01-02 0.004552 -0.995324 0.365204 5.000000 1.000000
2013-01-03 0.595855 1.133877 0.109665 5.000000 2.000000
2013-01-04 -0.178563 -1.369634 -0.562609 5.000000 3.000000
2013-01-05 -0.052989 1.368210 -1.233963 5.000000 4.000000
2013-01-06 1.062178 -0.798260 -0.237608 5.000000 5.000000
```

A `where` operation with setting:
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2

Out[54]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-0.325377</td>
<td>5.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>-0.365204</td>
<td>5.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.595855</td>
<td>-1.133877</td>
<td>-0.109665</td>
<td>5.0</td>
<td>-2.0</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
<td>-0.562609</td>
<td>5.0</td>
<td>-3.0</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.052989</td>
<td>-1.368210</td>
<td>-1.233963</td>
<td>5.0</td>
<td>-4.0</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>-1.062178</td>
<td>-0.798260</td>
<td>-0.237608</td>
<td>5.0</td>
<td>-5.0</td>
</tr>
</tbody>
</table>

2.1.4 Missing data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the Missing Data section.

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data:

In [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])

In [56]: df1.loc[dates[0] : dates[1], "E"] = 1

In [57]: df1

Out[57]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-0.325377</td>
<td>5.0</td>
<td>NaN</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>-0.365204</td>
<td>5.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>5.0</td>
<td>2.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
<td>-0.562609</td>
<td>5.0</td>
<td>3.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

To drop any rows that have missing data:

In [58]: df1.dropna(how="any")

Out[58]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>0.365204</td>
<td>5.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Filling missing data:

In [59]: df1.fillna(value=5)

Out[59]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-0.325377</td>
<td>5.0</td>
<td>5.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>0.365204</td>
<td>5.0</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.595855</td>
<td>1.133877</td>
<td>0.109665</td>
<td>5.0</td>
<td>2.0</td>
<td>5.0</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-0.178563</td>
<td>-1.369634</td>
<td>-0.562609</td>
<td>5.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

To get the boolean mask where values are nan:
2.1.5 Operations

See the Basic section on Binary Ops.

Stats

Operations in general exclude missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A   0.236988
B  -0.110189
C  -0.314115
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01  1.168656
2013-01-02  1.073066
2013-01-03  1.767879
2013-01-04  1.177839
2013-01-05  1.816252
2013-01-06  2.005262
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension:

```
In [63]: s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)
```

```
In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
```

(continues on next page)
Freq: D, dtype: float64

**In [65]:** df.sub(s, axis="index")

**Out[65]:**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.404145</td>
<td>0.133877</td>
<td>-0.890335</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>-3.178563</td>
<td>-4.369634</td>
<td>-3.562609</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-5.052989</td>
<td>-3.631790</td>
<td>-6.233963</td>
<td>0.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

**Apply**

Applying functions to the data:

**In [66]:** df.apply(np.cumsum)

**Out[66]:**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-0.325377</td>
<td>5</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>-0.004552</td>
<td>-0.995324</td>
<td>0.039827</td>
<td>10</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>0.591302</td>
<td>0.138553</td>
<td>0.149493</td>
<td>15</td>
<td>3.0</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.412739</td>
<td>-1.231082</td>
<td>-0.413117</td>
<td>20</td>
<td>6.0</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>0.359750</td>
<td>0.137129</td>
<td>-1.647079</td>
<td>25</td>
<td>10.0</td>
</tr>
<tr>
<td>2013-01-06</td>
<td>1.421927</td>
<td>-0.661132</td>
<td>-1.884687</td>
<td>30</td>
<td>15.0</td>
</tr>
</tbody>
</table>
```

**In [67]:** df.apply(\lambda x: x.max() - x.min())

**Out[67]:**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.240741</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2.737845</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.599166</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0.000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>4.000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

dtype: float64

**Histogramming**

See more at *Histogramming and Discretization*.

**In [68]:** s = pd.Series(np.random.randint(0, 7, size=10))

**In [69]:** s

**Out[69]:**

```
0    2
1    3
2    3
3    3
4    0
5    4
6    4
```

(continues on next page)
String Methods

Series is equipped with a set of string processing methods in the \texttt{str} attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in \texttt{str} generally uses regular expressions by default (and in some cases always uses them). See more at \textit{Vectorized String Methods}.

```

In [72]: s.str.lower()
Out[72]:
0    a
1    b
2    c
3   aaba
4   baca
5       NaN
6    caba
7    dog
8    cat
dtype: object
```

2.1.6 Merge

\textbf{Concat}

\texttt{pandas} provides various facilities for easily combining together \texttt{Series} and \texttt{DataFrame} objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the \textit{Merging section}.

Concatenating pandas objects together with \texttt{concat}:

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]:
   0     1     2     3
0 -1.58700 0.945705 -0.969482 1.359756
(continues on next page)
```
# Break it into pieces

```python
In [75]:
pieces = [df[:3], df[3:7], df[7:]]
```

```python
In [76]:
pd.concat(pieces)
```

```
Out[76]:
   0   1   2   3
0 -1.587000 -0.945705 -0.969482 1.359756
1 -1.556393 -0.878035 3.463794 2.501760
2  1.025253  0.321251 -0.033070  0.046852
3 -1.000667 -1.180260 -1.492427  0.489463
4 -0.238091  1.249817 -0.997295  0.588426
5 -1.267339 -0.697092 -0.339502  0.338748
6 -0.299343  1.547967  0.695675 -0.568623
7 -1.260831 -0.887644  1.079277  1.057432
8  0.273029  0.987568  0.964754  0.815525
9  1.365987  1.359370  0.268957 -0.209224
```

**Note:** Adding a column to a DataFrame is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the DataFrame constructor instead of building a DataFrame by iteratively appending records to it.

### Join

SQL style merges. See the Database style joining section.

```python
In [77]:
left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})
```

```python
In [78]:
right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})
```

```python
In [79]:
left
Out[79]:
  key  lval
0 foo    1
1 foo    2
```

```python
In [80]:
right
Out[80]:
  key  rval
0 foo    4
```
Another example that can be given is:

```
In [82]: left = pd.DataFrame({"key": ["foo", "bar"], "lval": [1, 2]})
In [83]: right = pd.DataFrame({"key": ["foo", "bar"], "rval": [4, 5]})
In [84]: left
Out[84]:
   key   lval
0   foo     1
1   bar     2
In [85]: right
Out[85]:
   key   rval
0   foo     4
1   bar     5
In [86]: pd.merge(left, right, on="key")
Out[86]:
   key   lval   rval
0   foo     1     4
1   bar     2     5
```

### 2.1.7 Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

See the *Grouping section*.

```
In [87]: df = pd.DataFrame(
    ....:     {  
    ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],  
    ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],  
    ....:         "C": np.random.randn(8),  
    ....:         "D": np.random.randn(8)  
    ....:     }
```

In [88]: df
Out[88]:
    A    B    C    D
0  foo  one  0.090398 -1.224360
1  bar  one -0.386444 -0.255209
2  foo  two  1.756941  0.072114
3  bar  three -1.193122  1.060455
4  foo  two -1.722343 -0.479172
5  bar  two -0.499584  0.376225
6  foo  one -0.204586 -1.520319
7  foo  three -0.743705 -0.730530

Grouping and then applying the `sum()` function to the resulting groups:

In [89]: df.groupby("A").sum()
Out[89]:
    C    D
A
bar -2.079150  1.181470
foo -0.823295 -3.882267

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum()` function:

In [90]: df.groupby(["A", "B"]).sum()
Out[90]:
     C     D
A B
bar one -0.386444 -0.255209
    three -1.193122  1.060455
    two -0.499584  0.376225
foo one -0.114188 -2.744679
    three -0.743705 -0.730530
    two  0.034598 -0.407058

2.1.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

Stack

In [91]: tuples = list(
    ....:     zip(
    ....:         ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
    ....:         ["one", "two", "one", "two", "one", "two", "one", "two"],
    ....:     )
    ....: )

(continues on next page)
In [92]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [93]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [94]: df2 = df[:4]

In [95]: df2
Out[95]:
    A     B
first second
bar one  -0.475880  1.022977
    two  -0.170081  -1.757496
baz one   0.881181   0.863468
    two  -0.302591  -0.061910

The `stack()` method “compresses” a level in the DataFrame’s columns:

In [96]: stacked = df2.stack()

In [97]: stacked
Out[97]:
    A     B
first second
bar one  -0.475880  1.022977
    two  -0.170081  -1.757496
baz one   0.881181   0.863468
    two  -0.302591  -0.061910
dtype: float64

With a “stacked” DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack()` is `unstack()`, which by default unstacks the last level:

In [98]: stacked.unstack()
Out[98]:
    A     B
first second
bar one  -0.475880  1.022977
    two  -0.170081  -1.757496
baz one   0.881181   0.863468
    two  -0.302591  -0.061910

In [99]: stacked.unstack(1)
Out[99]:
    second
first   one   two
bar A  -0.475880  -0.170081
    B   1.022977  1.757496
baz A   0.881181   0.863468
    B  -0.302591  -0.061910
Pivot tables

See the section on Pivot Tables.

```python
In [103]: pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
Out[103]:
   C    bar    foo
A
one    0.403664  0.231005
```

We can produce pivot tables from this data very easily:
2.1.9 Time series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the Time Series section.

```python
In [104]: rng = pd.date_range("1/1/2012", periods=100, freq="S")
In [105]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
In [106]: ts.resample("5Min").sum()
Out[106]:
2012-01-01 25490
Freq: 5T, dtype: int64
```

Time zone representation:

```python
In [107]: rng = pd.date_range("3/6/2012 00:00", periods=5, freq="D")
In [108]: ts = pd.Series(np.random.randn(len(rng)), rng)
In [109]: ts
Out[109]:
2012-03-06  1.412361
2012-03-07  0.244754
2012-03-08  0.105951
2012-03-09  0.243971
2012-03-10 -0.661610
Freq: D, dtype: float64
```

Converting to another time zone:

```python
In [110]: ts_utc = ts.tz_localize("UTC")
In [111]: ts_utc
Out[111]:
2012-03-06 00:00:00+00:00  1.412361
2012-03-07 00:00:00+00:00  0.244754
2012-03-08 00:00:00+00:00  0.105951
2012-03-09 00:00:00+00:00  0.243971
2012-03-10 00:00:00+00:00 -0.661610
Freq: D, dtype: float64
```

2.1. 10 minutes to pandas

167
Converting between time span representations:

```
In [113]: rng = pd.date_range("1/1/2012", periods=5, freq="M")

In [114]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [115]: ts
Out[115]:
2012-01-31 2.252741  
2012-02-29 1.607329  
2012-03-31 -0.263868  
2012-04-30 1.494371  
2012-05-31 1.406876  
Freq: M, dtype: float64
```

```
In [116]: ps = ts.to_period()

In [117]: ps
Out[117]:
2012-01  2.252741  
2012-02  1.607329  
2012-03 -0.263868  
2012-04  1.494371  
2012-05  1.406876  
Freq: M, dtype: float64
```

```
In [118]: ps.to_timestamp()
Out[118]:
2012-01-01  2.252741  
2012-02-01  1.607329  
2012-03-01 -0.263868  
2012-04-01  1.494371  
2012-05-01  1.406876  
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [119]: prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")

In [120]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [121]: ts.index = (prng.asfreq("M", "e") + 1).asfreq("H", "s") + 9
```

(continues on next page)
In [122]: ts.head()
Out[122]:
1990-03-01 09:00    -0.427020
1990-06-01 09:00    -0.013167
1990-09-01 09:00    -1.402264
1990-12-01 09:00     0.922385
1991-03-01 09:00     1.999645
Freq: H, dtype: float64

2.1.10 Categoricals

pandas can include categorical data in a DataFrame. For full docs, see the categorical introduction and the API documentation.

In [123]: df = pd.DataFrame(
        .....:     {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
        .....: )

Converting the raw grades to a categorical data type:

In [124]: df["grade"] = df["raw_grade"].astype("category")
In [125]: df["grade"]
Out[125]:
0  a
1  b
2  b
3  a
4  a
5  e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']

Rename the categories to more meaningful names (assigning to Series.cat.categories() is in place!):

In [126]: df["grade"].cat.categories = ["very good", "good", "very bad"]

Reorder the categories and simultaneously add the missing categories (methods under Series.cat() return a new Series by default):

In [127]: df["grade"] = df["grade"].cat.set_categories(
        .....:     ["very bad", "bad", "medium", "good", "very good"]
        .....: )

In [128]: df["grade"]
Out[128]:
0  very good
1     good
2 good
3 very good
4 very good
5 very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']

Sorting is per order in the categories, not lexical order:

```python
In [129]: df.sort_values(by="grade")
Out[129]:
   id  raw_grade  grade
  5   6           e  very bad
  1   2           b   good
  2   3           b   good
  0   1           a  very good
  3   4           a  very good
  4   5           a  very good
```

Grouping by a categorical column also shows empty categories:

```python
In [130]: df.groupby("grade").size()
Out[130]:
grade
very bad    1
  bad        0
  medium     0
  good       2
very good   3
dtype: int64
```

### 2.1.11 Plotting

See the `Plotting` docs.

We use the standard convention for referencing the matplotlib API:

```python
In [131]: import matplotlib.pyplot as plt
In [132]: plt.close("all")
```

The `close()` method is used to `close` a figure window:

```python
In [133]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
In [134]: ts = ts.cumsum()
In [135]: ts.plot();
```
If running under Jupyter Notebook, the plot will appear on `plt.show()`. Otherwise use `matplotlib.pyplot.show` to show it or `matplotlib.pyplot.savefig` to write it to a file.

```python
In [136]: plt.show();
```

On a DataFrame, the `plot()` method is a convenience to plot all of the columns with labels:

```python
In [137]: df = pd.DataFrame(
          .....:     np.random.randn(1000, 4), index=ts.index, columns=["A", "B", "C", "D"]
          .....:)
          .....:
In [138]: df = df.cumsum()
In [139]: plt.figure();
In [140]: df.plot();
In [141]: plt.legend(loc='best');
```
2.1.12 Getting data in/out

CSV

Writing to a csv file:

```
In [142]: df.to_csv("foo.csv")
```

Reading from a csv file:

```
In [143]: pd.read_csv("foo.csv")
Out[143]:
     Unnamed  0   A    B    C    D
0  2000-01-01 -0.546101 0.061245 -0.586319 -1.019486
1  2000-01-02  0.006307 -0.792411 -1.603888 -0.344875
2  2000-01-03 -0.401940 -1.678340 -1.085187  0.635465
3  2000-01-04 -0.722254 -1.382689 -1.999852 -0.110589
4  2000-01-05 -1.012887 -1.159195 -2.185900  0.638026
   ...     ... ... ... ... ...
995 2002-09-22  7.881616 37.041776  5.819321  78.464831
996 2002-09-23 10.660594 37.264346  4.826818  81.082512
997 2002-09-24 10.736323 35.703505  2.645408  81.240208
```
HDF5

Reading and writing to HDFStores.

Writing to a HDF5 Store:

```
In [144]: df.to_hdf("foo.h5", "df")
```

Reading from a HDF5 Store:

```
In [145]: pd.read_hdf("foo.h5", "df")
```

```text
Out[145]:
   A          B          C          D
0 2000-01-01 -0.546101  0.061245 -0.586319 -1.019486
1 2000-01-02  0.006307 -0.792411 -1.603888 -0.348753
2 2000-01-03 -0.401940 -1.678340 -1.085187  0.635465
3 2000-01-04 -0.722254 -1.382689 -1.999852 -0.110589
4 2000-01-05 -1.012887 -1.159195 -2.185900  0.638026
   ... ... ... ... ...
899 2002-09-22  7.881616  37.041776  5.819321  78.464831
900 2002-09-23 10.660594  37.264346  4.826818  81.082512
901 2002-09-24 10.736323  35.703505  2.645408  81.240208
902 2002-09-25 11.787296  36.695909  3.445171  79.737522
903 2002-09-26 12.157068  36.695909  3.209820  80.149341
```

[1000 rows x 4 columns]

Excel

Reading and writing to MS Excel.

Writing to an excel file:

```
In [146]: df.to_excel("foo.xlsx", sheet_name="Sheet1")
```

Reading from an excel file:

```
In [147]: pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA")
```

```text
Out[147]:
      Unnamed: 0      A      B      C      D
0  2000-01-01     -0.5461  0.0612 -0.5863 -1.0195
1  2000-01-02     0.0063 -0.7924 -1.6039 -0.3488
2  2000-01-03    -0.4019 -1.6783 -1.0852  0.6355
3  2000-01-04   -0.7223 -1.3827 -1.9999 -0.1106
4  2000-01-05   -1.0129 -1.1592 -2.1859  0.6380
   ...      ...      ...      ...      ...
899 2002-09-22   7.8816  37.0418  5.8193  78.4648
900 2002-09-23  10.6606  37.2643  4.8268  81.0825
901 2002-09-24  10.7363  35.7035  2.6454  81.2402
902 2002-09-25  11.7873  36.6959  3.4452  79.7375
903 2002-09-26  12.1571  36.6959  3.2098  80.1493
```

[1000 rows x 4 columns]

2.1. 10 minutes to pandas
2.1.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

```python
>>> if pd.Series([False, True, False]):
    ...    print("I was true")
Traceback
...    ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *Comparisons* for an explanation and what to do.

See *Gotchas* as well.

2.2 Intro to data structures

We’ll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

```
In [1]: import numpy as np
In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic.** The link between labels and data will not be broken unless done so explicitly by you.

We’ll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

2.2.1 Series

*Series* is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index.** The basic method to create a Series is to call:

```python
>>> s = pd.Series(data, index=index)
```

Here, *data* can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed `index` is a list of axis labels. Thus, this separates into a few cases depending on what `data` is:

### From ndarray

If `data` is an ndarray, `index` must be the same length as `data`. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```python
In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [4]: s
Out[4]:
   a    0.469112
   b   -0.282863
   c   -1.509059
   d   -1.135632
   e    1.212112
 dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='O')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
   0   -0.173215
   1    0.119209
   2   -1.044236
   3   -0.861849
   4   -2.104569
 dtype: float64
```

**Note:** pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

### From dict

Series can be instantiated from dicts:

```python
In [7]: d = {'b': 1, 'a': 0, 'c': 2}

In [8]: pd.Series(d)
Out[8]:
   b    1
   a    0
   c    2
dtype: int64
```

**Note:** When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict’s insertion order, if you’re using Python version >= 3.6 and pandas version >= 0.23.

If you’re using Python < 3.6 or pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered
list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. ['a', 'b', 'c'] rather than ['b', 'a', 'c']).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {"a": 0.0, "b": 1.0, "c": 2.0}
In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64
In [11]: pd.Series(d, index=["b", "c", "d", "a"])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

**Note:** NaN (not a number) is the standard missing data marker used in pandas.

### From scalar value

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of `index`.

```
In [12]: pd.Series(5.0, index=["a", "b", "c", "d", "e"])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

### Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.4691122999071863

In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c  -1.509059
```

(continues on next page)
dtype: float64

In [15]: s[s > s.median()]
Out[15]:
da 0.469112
e 1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e -1.135632
d -0.282863
b 0.282863
dtype: float64

In [17]: np.exp(s)
Out[17]:
a 1.598575
b 0.753623
c 0.221118
d 0.321219
e 3.360575
dtype: float64

Note: We will address array-based indexing like s[[4, 3, 1]] in section on indexing.

Like a NumPy array, a pandas Series has a dtype.

In [18]: s.dtype
Out[18]: dtype('float64')

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy’s type system in a few places, in which case the dtype would be an ExtensionDtype. Some examples within pandas are Categorical data and Nullable integer data type. See dtypes for more.

If you need the actual array backing a Series, use Series.array.

In [19]: s.array
Out[19]: <PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
-1.1356323710171934, 1.2121120250208506]
Length: 5, dtype: float64

Accessing the array can be useful when you need to do some operation without the index (to disable automatic alignment, for example).

Series.array will always be an ExtensionArray. Briefly, an ExtensionArray is a thin wrapper around one or more concrete arrays like a numpy.ndarray. pandas knows how to take an ExtensionArray and store it in a Series or a column of a DataFrame. See dtypes for more.

While Series is ndarray-like, if you need an actual ndarray, then use Series.to_numpy().

2.2. Intro to data structures 177
Even if the Series is backed by a `ExtensionArray`, `Series.to_numpy()` will return a NumPy ndarray.

**Series is dict-like**

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s["a"]
Out[21]: 0.469112

In [22]: s["e"] = 12.0

In [23]: s
Out[23]:
   a    0.469112
   b   -0.282863
   c   -1.509059
   d   -1.135632
   e    12.000000
      dtype: float64

In [24]: "e" in s
Out[24]: True

In [25]: "f" in s
Out[25]: False
```

If a label is not contained, an exception is raised:

```
>>> s["f"]
KeyError: 'f'
```

Using the `get` method, a missing label will return `None` or specified default:

```
In [26]: s.get("f")
```

```
In [27]: s.get("f", np.nan)
Out[27]: nan
```

See also the `section on attribute access`.

**Vectorized operations and label alignment with Series**

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a    0.938225
b   -0.565727
```

(continues on next page)
A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

The result of an operation between unaligned Series will have the `union` of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing `NaN`. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

**Note:** In general, we chose to make the default result of operations between differently indexed objects yield the `union` of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the `dropna` function.
Name attribute

Series can also have a name attribute:

```python
In [32]: s = pd.Series(np.random.randn(5), name="something")

In [33]: s
Out[33]:
0  -0.494929
1   1.071804
2   0.721555
3  -0.706771
4  -1.039575
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

The Series name will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

You can rename a Series with the `pandas.Series.rename()` method.

```python
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that s and s2 refer to different objects.

### 2.2.2 DataFrame

**DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass `index` (row labels) and `columns` (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

**Note:** When the data is a dict, and columns is not specified, the DataFrame columns will be ordered by the dict’s insertion order, if you are using Python version >= 3.6 and pandas >= 0.23.

If you are using Python < 3.6 or pandas < 0.23, and columns is not specified, the DataFrame columns will be the lexically ordered list of dict keys.
From dict of Series or dicts

The resulting index will be the union of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {
        ....:   "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"],
        ....:   "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
        ....: }
        ....:

In [38]: df = pd.DataFrame(d)

In [39]: df
Out[39]:
   one  two
  a  1.0  1.0
  b  2.0  2.0
  c  3.0  3.0
  d   NaN  4.0

In [40]: pd.DataFrame(d, index=["d", "b", "a"])
Out[40]:
   one  two
  d   NaN  4.0
  b  2.0   NaN
  a  1.0   NaN

In [41]: pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
Out[41]:
   two  three
  d    4.0   NaN
  b  2.0    NaN
  a  1.0    NaN
```

The row and column labels can be accessed respectively by accessing the index and columns attributes:

```
Note: When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```
From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```python
In [44]: d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0   1.0  4.0
1   2.0  3.0
2   3.0  2.0
3   4.0  1.0

In [46]: pd.DataFrame(d, index=["a", "b", "c", "d")
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```python
In [47]: data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
In [48]: data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]
In [49]: pd.DataFrame(data)
Out[49]:
   A     B         C
0  1.0  2.0 'Hello'
1  2.0  3.0 'World'

In [50]: pd.DataFrame(data, index=["first", "second")
Out[50]:
   A     B         C
first  1.0  2.0 'Hello'
second  2.0  3.0 'World'

In [51]: pd.DataFrame(data, columns=["C", "A", "B")
Out[51]:
   C     A     B
0 'Hello' 1  2.0
1 'World' 2  3.0
```

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.
From a list of dicts

In [52]: data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]

In [53]: pd.DataFrame(data2)
Out[53]:
   a  b  c
0  1  2 NaN
1  5 10  20.0

In [54]: pd.DataFrame(data2, index=["first", "second"])
Out[54]:
   a  b  c
first 1  2 NaN
second 5 10  20.0

In [55]: pd.DataFrame(data2, columns=["a", "b"])
Out[55]:
   a  b
0  1  2
1  5 10

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

In [56]: pd.DataFrame(
    .....:    {
    .....:    ("a", "b") : {("A", "B") : 1, ("A", "C") : 2},
    .....:    ("a", "a") : {("A", "C") : 3, ("A", "B") : 4},
    .....:    ("a", "c") : {("A", "B") : 5, ("A", "C") : 6},
    .....:    ("b", "a") : {("A", "C") : 7, ("A", "B") : 8},
    .....:    ("b", "b") : {("A", "D") : 9, ("A", "B") : 10},
    .....:    }
    .....: )
Out[56]:
   a  b
   b a c    a  b
A B 1.0 4.0 5.0 8.0 10.0
C 2.0 3.0 6.0 7.0 NaN
D NaN NaN NaN NaN 9.0
**From a Series**

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

**From a list of namedtuples**

The field names of the first namedtuple in the list determine the columns of the DataFrame. The remaining namedtuples (or tuples) are simply unpacked and their values are fed into the rows of the DataFrame. If any of those tuples is shorter than the first namedtuple then the later columns in the corresponding row are marked as missing values. If any are longer than the first namedtuple, a ValueError is raised.

```
In [57]: from collections import namedtuple
In [58]: Point = namedtuple("Point", "x y")
In [59]: pd.DataFrame([Point(0, 0), Point(0, 3), (2, 3)])
Out[59]:
      x  y
0    0  0
1    0  3
2    2  3
In [60]: Point3D = namedtuple("Point3D", "x y z")
In [61]: pd.DataFrame([Point3D(0, 0, 0), Point3D(0, 3, 5), Point(2, 3)])
Out[61]:
      x  y  z
0    0  0  0.0
1    0  3  5.0
2    2  3  NaN
```

**From a list of dataclasses**

New in version 1.1.0.

Data Classes as introduced in PEP557, can be passed into the DataFrame constructor. Passing a list of dataclasses is equivalent to passing a list of dictionaries.

Please be aware, that all values in the list should be dataclasses, mixing types in the list would result in a TypeError.

```
In [62]: from dataclasses import make_dataclass
In [63]: Point = make_dataclass("Point", [("x", int), ("y", int)])
In [64]: pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
Out[64]:
      x  y
0    0  0
1    0  3
2    2  3
```

**Missing data**
Much more will be said on this topic in the Missing data section. To construct a DataFrame with missing data, we use \texttt{np.nan} to represent missing values. Alternatively, you may pass a \texttt{numpy.MaskedArray} as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

**Alternate constructors**

\texttt{DataFrame.from_dict} takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the DataFrame constructor except for the \texttt{orient} parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

\begin{verbatim}
In [65]: pd.DataFrame.from_dict(dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]))
Out[65]:
   A  B
0  1  4
1  2  5
2  3  6
\end{verbatim}

If you pass \texttt{orient='index'}, the keys will be the row labels. In this case, you can also pass the desired column names:

\begin{verbatim}
In [66]: pd.DataFrame.from_dict(
    ...:     dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]),
    ...:     orient="index",
    ...:     columns=["one", "two", "three"],
    ...: )
Out[66]:
      one  two  three
A  1  2.0  3
B  4  5.0  6
\end{verbatim}

\texttt{DataFrame.from_records} takes a list of tuples or an ndarray with structured dtypes. It works analogously to the normal DataFrame constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

\begin{verbatim}
In [67]: data
Out[67]:
array([[1, 2., b'Hello'], [2, 3., b'World']],
       dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [68]: pd.DataFrame.from_records(data, index="C")
Out[68]:
   A  B
C b'Hello'  1  2.0
b'World'  2  3.0
\end{verbatim}
Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [69]: df["one"]
Out[69]:
a  1.0
b  2.0
c  3.0
d NaN
Name: one, dtype: float64
```
```
In [70]: df["three"] = df["one"] * df["two"]
In [71]: df["flag"] = df["one"] > 2
In [72]: df
Out[72]:
   one  two  three  flag
  a  1.0  1.0  1.0  False
  b  2.0  2.0  4.0  False
  c  3.0  3.0  9.0   True
  d  NaN  4.0 NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [73]: del df["two"]
In [74]: three = df.pop("three")
In [75]: df
Out[75]:
   one  flag
  a 1.0 False
  b 2.0 False
  c 3.0 True
  d NaN False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [76]: df["foo"] = "bar"
In [77]: df
Out[77]:
   one  flag  foo
  a 1.0  False  bar
  b 2.0  False  bar
  c 3.0   True  bar
  d NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:
In [78]: df["one_trunc"] = df["one"][2]

In [79]: df
Out[79]:
   one  flag  foo  one_trunc
  a  1.0    False  bar   1.0
  b  2.0    False  bar   2.0
  c  3.0     True  bar   NaN
  d  NaN     False  bar   NaN

You can insert raw ndarrays but their length must match the length of the DataFrame’s index.

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns:

In [80]: df.insert(1, "bar", df["one"])  

In [81]: df
Out[81]:
   one  flag  foo  one_trunc  bar
  a  1.0    False  bar   1.0   1.0
  b  2.0    False  bar   2.0   2.0
  c  3.0     True  bar   NaN   NaN
  d  NaN     False  bar   NaN   NaN

Assigning new columns in method chains

Inspired by dplyr’s mutate verb, DataFrame has an assign() method that allows you to easily create new columns that are potentially derived from existing columns.

In [82]: iris = pd.read_csv("data/iris.data")

In [83]: iris.head()
Out[83]:
   SepalLength  SepalWidth  PetalLength  PetalWidth  Name  
 0         5.1         3.5         1.4       0.2  Iris-setosa  
 1         4.9         3.0         1.4       0.2  Iris-setosa  
 2         4.7         3.2         1.3       0.2  Iris-setosa  
 3         4.6         3.1         1.5       0.2  Iris-setosa  
 4         5.0         3.6         1.4       0.2  Iris-setosa  

In [84]: iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
Out[84]:
   SepalLength  SepalWidth  PetalLength  PetalWidth  Name  sepal_ratio  
 0         5.1         3.5         1.4       0.2  Iris-setosa    0.686275  
 1         4.9         3.0         1.4       0.2  Iris-setosa    0.612245  
 2         4.7         3.2         1.3       0.2  Iris-setosa    0.680851  
 3         4.6         3.1         1.5       0.2  Iris-setosa    0.673913  
 4         5.0         3.6         1.4       0.2  Iris-setosa    0.720000  

In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

2.2. Intro to data structures
assign always returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don’t have a reference to the DataFrame at hand. This is common when using assign in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```python
In [86]: (  
......:     iris.query("SepalLength > 5")  
......:         .assign(  
......:             SepalRatio= lambda x: x.SepalWidth / x.SepalLength,  
......:             PetalRatio= lambda x: x.PetalWidth / x.PetalLength,  
......:         )  
......:         .plot(kind="scatter", x="SepalRatio", y="PetalRatio")  
......: )
```

```
Out[86]: <AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```
Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that’s been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn’t have a reference to the filtered DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the DataFrame. A copy of the original DataFrame is returned, with the new values inserted.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for dependent assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```
In [87]: dfa = pd.DataFrame( { "A": [1, 2, 3], "B": [4, 5, 6] } )
In [88]: dfa.assign(C=lambda x: x["A"] + x["B"], D=lambda x: x["A"] + x["C"])
Out[88]:
          A     B     C     D
0  1.000  5.000  6.000  12.00
1  2.000  7.000  9.000  16.00
2  3.000  9.000  12.000 21.00
```

In the second expression, `x[‘C’]` will refer to the newly created column, that’s equal to `dfa[‘A’] + dfa[‘B’]`.  

2.2. Intro to data structures 189
Indexing / selection

The basics of indexing are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select column</td>
<td>df[col]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by label</td>
<td>df.loc[label]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by integer location</td>
<td>df.iloc[loc]</td>
<td>Series</td>
</tr>
<tr>
<td>Slice rows</td>
<td>df[5:10]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Select rows by boolean vector</td>
<td>df[bool_vec]</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```python
In [89]: df.loc["b"]
Out[89]:
one  2.0
bar  2.0
flag False
foo  bar
one_trunc 2.0
Name: b, dtype: object
```

In [90]: df.iloc[2]
Out[90]:
one  3.0
bar  3.0
flag True
foo  bar
one_trunc NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the section on indexing. We will address the fundamentals of reindexing / conforming to new sets of labels in the section on reindexing.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on both the columns and the index (row labels). Again, the resulting object will have the union of the column and row labels.

```python
In [91]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
In [92]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
In [93]: df + df2
Out[93]:
   A         B         C         D
0  0.045691 -0.014138  1.380871  NaN
1 -0.955398  1.501007  0.037181  NaN
2 -0.662690  1.534833  0.859691  NaN
3 -2.452949  1.237274 -0.133712  NaN
4  1.414490  1.951676  2.320422  NaN
5 -0.494922 -1.649727 -1.084601  NaN
   ...
5  1.414490  1.951676  2.320422  NaN
5 -0.494922 -1.649727 -1.084601  NaN
(continues on next page)
```
When doing an operation between DataFrame and Series, the default behavior is to align the Series index on the DataFrame columns, thus broadcasting row-wise. For example:

```
In [94]: df - df.iloc[0]
Out[94]:
     A     B     C     D
0  0.000000 0.000000 0.000000 0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128 0.829678 0.010026 -1.991234
3 -1.311128 0.054325 -1.724913 -1.620544
4  0.573025 1.500742 -0.676070 1.367331
5 -1.741248 0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282 0.000430
7 -0.743894 0.411013 -0.929563 -0.282386
8 -1.194921 1.320690 0.238224 -1.482644
9  2.293786 1.856228 0.773289 -1.46531
```

For explicit control over the matching and broadcasting behavior, see the section on flexible binary operations.

Operations with scalars are just as you would expect:

```
In [95]: df * 5 + 2
Out[95]:
     A     B     C     D
0  3.359299 -0.124862 4.835102 3.381160
1 -3.437003 -1.368449 2.568242 -5.392133
2  4.624938 4.023526 4.885230 -6.575010
3 -3.196342 0.146766 -3.789461 -4.721559
4  6.224426 7.378849 1.454750 10.217815
5 -5.346940 3.785103 -1.373001 -6.884519
6 -2.844569 -4.472618 4.068691 3.383309
7 -0.360173 -2.697986 -0.863638 -0.743875
8 -2.615303 6.478587 6.026220 -4.032059
9  14.828230 9.156280 8.701544 -3.851494
```

```
In [96]: 1 / df
Out[96]:
     A     B     C     D
0  3.678365 -2.353094 1.763605 3.620145
1 -0.919624 -1.483463 8.799067 -0.676395
2  1.964887 2.470934 1.732964 -0.583099
3 -0.962215 -2.697986 -0.863638 -0.743875
4  1.183593 0.929567 -9.170108 0.608434
5 -0.680555 2.800959 -1.482360 -0.562777
6 -1.032084 -0.772485 2.416988 3.614523
7 -2.118489 -71.634509 -2.758294 -162.507295
8 -1.083352 1.116424 1.241860 -0.828964
9  0.389765 0.698687 0.746097 -0.854483
```
In [97]: df ** 4
Out[97]:
A     B               C               D
0 0.005462 3.261689e-02 0.103370 5.822320e-03
1 1.398165 2.059869e-01 0.000167 4.777482e+00
2 0.075962 2.682596e-02 0.110877 8.650845e+00
3 1.166571 1.887302e-02 1.797515 3.265879e+00
4 0.509555 1.339298e+00 0.000141 7.297019e+00
5 4.661717 1.624699e-02 0.207103 9.969092e+00
6 0.881334 2.808277e+00 0.029302 5.858632e-03
7 0.049647 3.797614e-08 0.017276 1.433866e-09
8 43.329821 4.196326e+00 3.227153 1.875802e+00

Boolean operators work as well:
In [98]: df1 = pd.DataFrame({"a": [1, 0, 1], "b": [0, 1, 1]}, dtype=bool)
In [99]: df2 = pd.DataFrame({"a": [0, 1, 1], "b": [1, 1, 0]}, dtype=bool)
In [100]: df1 & df2
Out[100]:
a  b
0 False False
1 False True
2 True False
In [101]: df1 | df2
Out[101]:
a  b
0 True True
1 True True
2 True True
In [102]: df1 ^ df2
Out[102]:
a  b
0 True True
1 True False
2 False True
In [103]: -df1
Out[103]:
a  b
0 False True
1 True False
2 False False
Transposing

To transpose, access the T attribute (also the `transpose` function), similar to an ndarray:

```python
# only show the first 5 rows
In [104]: df[:5].T
Out[104]:
      0  1  2  3  4
A  0.27186 1.087401 0.524988 -1.039268 0.844885
B -0.424972 -0.673690 0.404705 -0.370647 1.075770
C  0.567020 0.113648 0.577046 -1.157892 -0.109050
D  0.276232 -1.478427 -1.715002 -1.344312 1.643563
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```python
In [105]: np.exp(df)
Out[105]:
     A    B     C    D
0  1.3124  0.6538  1.7630  1.3181
1  0.3371  0.5098  1.1204  0.2279
2  1.6904  1.4988  1.7807  0.1799
3  0.3537  0.6903  0.3142  0.2607
4  2.3277  2.9323  1.0866  5.1735
5  0.2301  1.4291  0.5093  0.1691
6  0.3795  0.2741  1.5124  1.3187
7  0.6237  0.9861  0.6959  0.9938
8  0.3973  2.4491  2.2372  0.2993
9  13.0109  4.1839  3.8202  0.3103
```

```python
In [106]: np.asarray(df)
Out[106]:
array([[ 0.2719, -0.425 , 0.567 , 0.2762],
       [-1.0874, -0.6737, 0.1136, -1.4784],
       [ 0.525 , 0.4047, 0.577 , -1.715 ],
       [-1.0393, -0.3706, -1.1579, -1.3443],
       [ 0.8449, 1.0758, -0.109 , 1.6436],
       [-1.4694, 0.357 , -0.6746, -1.7769],
       [-0.9689, -1.2945, 0.4137, 0.2767],
       [-0.472 , -0.014 , -0.3625, -0.0062],
       [-0.9231, 0.8957, 0.8052, -1.2064],
       [ 2.5656, 1.4313, 1.3403, -1.1703]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

`Series` implements `__array_ufunc__`, which allows it to work with NumPy’s universal functions.

The ufunc is applied to the underlying array in a Series.

```python
In [107]: ser = pd.Series([1, 2, 3, 4])
```

(continues on next page)
In [108]: `np.exp(ser)`
   Out[108]:
   0   2.718282
   1   7.389056
   2  20.085537
   3  54.598150
   dtype: float64

Changed in version 0.25.0: When multiple Series are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using `numpy.remainder()` on two Series with differently ordered labels will align before the operation.

In [109]: `ser1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])`
   In [110]: `ser2 = pd.Series([1, 3, 5], index=['b', 'a', 'c'])`
   In [111]: `ser1`
   Out[111]:
   a   1
   b   2
   c   3
   dtype: int64
   In [112]: `ser2`
   Out[112]:
   b   1
   a   3
   c   5
   dtype: int64
   In [113]: `np.remainder(ser1, ser2)`
   Out[113]:
   a   1
   b   0
   c   3
   dtype: int64

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

In [114]: `ser3 = pd.Series([2, 4, 6], index=['b', 'c', 'd'])`
   In [115]: `ser3`
   Out[115]:
   b   2
   c   4
   d   6
   dtype: int64
   In [116]: `np.remainder(ser1, ser3)`
   Out[116]:
   a   NaN
   (continues on next page)
When a binary ufunc is applied to a Series and Index, the Series implementation takes precedence and a Series is returned.

```
In [117]: ser = pd.Series([1, 2, 3])
In [118]: idx = pd.Index([4, 5, 6])
In [119]: np.maximum(ser, idx)
Out[119]:
    0  4
    1  5
    2  6
```

NumPy ufuncs are safe to apply to Series backed by non-ndarray arrays, for example arrays.SparseArray (see Sparse calculation). If possible, the ufunc is applied without converting the underlying data to an ndarray.

### Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using info(). (Here I am reading a CSV version of the baseball dataset from the plyr R package):

```
In [120]: baseball = pd.read_csv("data/baseball.csv")
```

```
In [121]: print(baseball)
   id  player   year  stint  team  lg  g  ab  r  h  X2b  X3b  hr  rbi  sb  cs  bb  so  ibb  hbp  sh  sf  gidp
---  ------  ----  ----  ----  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  ---
 0  88641  womacto01  2006  2  CHN  NL  19  50  6  14  1  0  1  2.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0
 1  88643  schilcu01  2006  1  BOS  AL  31  2  0  1  0  0  0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
98  89533  aloumo01  2007  1  NYN  NL  87  328  51  112  19  1  13  49.0  3.0  0.0  0.0  0.0  0.0  0.0
99  89534  alomasa02  2007  1  NYN  NL  8  22  1  3  1  0  0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

[100 rows x 23 columns]

```
In [122]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
   # Column  Non-Null Count  Dtype
   ---  -------  -----------  ----
   0   id      100 non-null  int64
```

(continues on next page)
However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won’t always fit the console width:

```
In [123]: print(baseball.iloc[-20:, :12].to_string())
```

<table>
<thead>
<tr>
<th>id</th>
<th>player</th>
<th>year</th>
<th>stint</th>
<th>team</th>
<th>lg</th>
<th>g</th>
<th>ab</th>
<th>r</th>
<th>h</th>
<th>X2b</th>
<th>X3b</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>finlest01</td>
<td>2007</td>
<td>1</td>
<td>COL</td>
<td>NL</td>
<td>43</td>
<td>94</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>81</td>
<td>embreal01</td>
<td>2007</td>
<td>1</td>
<td>OAK</td>
<td>AL</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>82</td>
<td>edmonji01</td>
<td>2007</td>
<td>1</td>
<td>SLN</td>
<td>NL</td>
<td>117</td>
<td>365</td>
<td>39</td>
<td>92</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>83</td>
<td>easleda01</td>
<td>2007</td>
<td>1</td>
<td>NYN</td>
<td>NL</td>
<td>76</td>
<td>193</td>
<td>24</td>
<td>54</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>delgaca01</td>
<td>2007</td>
<td>1</td>
<td>NYN</td>
<td>NL</td>
<td>139</td>
<td>538</td>
<td>71</td>
<td>139</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>cormirh01</td>
<td>2007</td>
<td>1</td>
<td>CIN</td>
<td>NL</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>coninje01</td>
<td>2007</td>
<td>2</td>
<td>NYN</td>
<td>NL</td>
<td>21</td>
<td>41</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>coninje01</td>
<td>2007</td>
<td>1</td>
<td>CIN</td>
<td>NL</td>
<td>80</td>
<td>215</td>
<td>23</td>
<td>57</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>88</td>
<td>clemero02</td>
<td>2007</td>
<td>1</td>
<td>NYA</td>
<td>AL</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>89</td>
<td>claytro01</td>
<td>2007</td>
<td>2</td>
<td>BOS</td>
<td>AL</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>claytro01</td>
<td>2007</td>
<td>1</td>
<td>TOR</td>
<td>AL</td>
<td>69</td>
<td>189</td>
<td>23</td>
<td>48</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>cirilje01</td>
<td>2007</td>
<td>2</td>
<td>ARI</td>
<td>NL</td>
<td>28</td>
<td>40</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>92</td>
<td>cirilje01</td>
<td>2007</td>
<td>1</td>
<td>MIN</td>
<td>AL</td>
<td>50</td>
<td>153</td>
<td>18</td>
<td>40</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>93</td>
<td>bondsba01</td>
<td>2007</td>
<td>1</td>
<td>SFN</td>
<td>NL</td>
<td>126</td>
<td>340</td>
<td>75</td>
<td>94</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>94</td>
<td>biggcrc01</td>
<td>2007</td>
<td>1</td>
<td>HOU</td>
<td>NL</td>
<td>141</td>
<td>517</td>
<td>68</td>
<td>130</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>95</td>
<td>benitar01</td>
<td>2007</td>
<td>2</td>
<td>FLO</td>
<td>NL</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>benitar01</td>
<td>2007</td>
<td>1</td>
<td>SFN</td>
<td>NL</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>ausmubr01</td>
<td>2007</td>
<td>1</td>
<td>HOU</td>
<td>NL</td>
<td>117</td>
<td>349</td>
<td>38</td>
<td>82</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>98</td>
<td>aloumo01</td>
<td>2007</td>
<td>1</td>
<td>NYN</td>
<td>NL</td>
<td>87</td>
<td>328</td>
<td>51</td>
<td>112</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>99</td>
<td>alomasa02</td>
<td>2007</td>
<td>1</td>
<td>NYN</td>
<td>NL</td>
<td>8</td>
<td>22</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Wide DataFrames will be printed across multiple rows by default:
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [124]: pd.DataFrame(np.random.randn(3, 12))
Out[124]:
    0    1    2    3    4    5    6    7    8    9    10   11
0 -1.226825 0.769804 -1.281247 -0.727707 -0.097883 0.695775 0.341734 0.
   -0.959726 -1.110336 -0.619976 0.149748
1 -0.732339 0.687738 0.176444 0.403310 -0.154951 0.301624 -2.179861 -1.369849 -0.
   -0.954208 1.462696 -1.743161 -0.826591
2 -0.345352 1.314232 0.690579 0.995761 2.396780 0.014871 3.357427 -0.317441 -1.
   -2.36269 0.896171 -0.487602 -0.082240

You can change how much to print on a single row by setting the display.width option:

In [125]: pd.set_option("display.width", 40)  # default is 80

In [126]: pd.DataFrame(np.random.randn(3, 12))
Out[126]:
    0    1    2    3    4    5    6    7    8    9    10   11
0 -2.182937 0.380396 0.084844 0.432390 1.519970 -0.493662 0.600178 0.274230 0.
   -0.32885 -0.023688 2.410179 1.450520
1 0.206053 -0.251905 -2.213588 1.063327 1.266143 -0.863838 0.408204 -1.
   -0.48089 -0.025747 -0.988387 0.094055
2 1.262731 1.289997 0.082423 -0.055758 0.536580 -0.489682 0.369374 -0.034571 -2.
   -0.484478 -0.281461 0.030711 0.109121

You can adjust the max width of the individual columns by setting display.max_colwidth

In [127]: datafile = {
   ......:   "filename": ["filename_01", "filename_02"],
   ......:   "path": [
   ......:     "media/user_name/storage/folder_01/filename_01",
   ......:     "media/user_name/storage/folder_02/filename_02",
   ......:   ],
   ......: }

In [128]: pd.set_option("display.max_colwidth", 30)

In [129]: pd.DataFrame(datafile)
Out[129]:
   filename   path
0 filename_01 media/user_name/storage/folder_01/filename_01
1 filename_02 media/user_name/storage/folder_02/filename_02

In [130]: pd.set_option("display.max_colwidth", 100)

In [131]: pd.DataFrame(datafile)
Out[131]:
   filename   path
0 filename_01 media/user_name/storage/folder_01/filename_01
1 filename_02 media/user_name/storage/folder_02/filename_02

You can also disable this feature via the expand_frame_repr option. This will print the table in one block.

2.2. Intro to data structures  197
DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```python
In [132]: df = pd.DataFrame({"foo1": np.random.randn(5), "foo2": np.random.randn(5)})

In [133]: df
Out[133]:
   foo1    foo2
0  1.126203  0.781836
1 -0.977349 -1.071357
2  1.474071  0.441153
3 -0.064034  2.353925
4 -1.282782  0.583787

In [134]: df.foo1
Out[134]:
0   1.126203
1  -0.977349
2   1.474071
3  -0.064034
4  -1.282782
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```python
In [5]: df.foo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```

2.3 Essential basic functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. To begin, let’s create some example objects like we did in the 10 minutes to pandas section:

```python
In [1]: index = pd.date_range("1/1/2000", periods=8)
In [2]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e")
In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C")
```

2.3.1 Head and tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```python
In [4]: long_series = pd.Series(np.random.randn(1000))
In [5]: long_series.head()
Out[5]:
0  -1.157892
```

(continues on next page)
2.3.2 Attributes and underlying data

pandas objects have a number of attributes enabling you to access the metadata

- `shape`: gives the axis dimensions of the object, consistent with ndarray

- **Axis labels**
  - **Series**: `index` (only axis)
  - **DataFrame**: `index` (rows) and `columns`

Note, these attributes can be safely assigned to!

```python
In [7]: df[:2]
Out[7]:
          A         B         C
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929

In [8]: df.columns = [x.lower() for x in df.columns]

In [9]: df
Out[9]:
         a         b         c
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929
2000-01-03  1.071804  0.721555 -0.706771
2000-01-04 -1.039575  0.271860  0.424972
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427
2000-01-07  0.524988  0.404705  0.577046
2000-01-08 -1.715002 -1.039268 -0.370647
```

pandas objects (`Index`, `Series`, `DataFrame`) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may extend NumPy’s type system to add support for custom arrays (see `dtypes`).

To get the actual data inside a `Index` or `Series`, use the `.array` property

2.3. Essential basic functionality
array will always be an ExtensionArray. The exact details of what an ExtensionArray is and why pandas uses them are a bit beyond the scope of this introduction. See dtypes for more.

If you know you need a NumPy array, use to_numpy() or numpy.asarray().

When the Series or Index is backed by an ExtensionArray, to_numpy() may involve copying data and coercing values. See dtypes for more.

to_numpy() gives some control over the dtype of the resulting numpy.ndarray. For example, consider datetimes with timezones. NumPy doesn’t have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype numpy.ndarray with Timestamp objects, each with the correct tz
2. A datetime64[ns]-dtype numpy.ndarray, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with dtype=object

Getting the “raw data” inside a DataFrame is possibly a bit more complex. When your DataFrame only has a single data type for all the columns, DataFrame.to_numpy() will return the underlying data:
If a DataFrame contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame’s columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

**Note:** When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

In the past, pandas recommended `Series.values` or `DataFrame.values` for extracting the data from a Series or DataFrame. You’ll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an extension type, it’s unclear whether `Series.values` returns a NumPy array or the extension array. `Series.array` will always return an `ExtensionArray`, and will never copy data. `Series.to_numpy()` will always return a NumPy array, potentially at the cost of copying / coercing values.

2. When your DataFrame contains a mixture of data types, `DataFrame.values` may involve copying data and coercing values to a common dtype, a relatively expensive operation. `DataFrame.to_numpy()`, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

### 2.3.3 Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have nans.

Here is a sample (using 100 column x 100,000 row DataFrames):

<table>
<thead>
<tr>
<th>Operation</th>
<th>0.11.0 (ms)</th>
<th>Prior Version (ms)</th>
<th>Ratio to Prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>df1 &gt; df2</td>
<td>13.32</td>
<td>125.35</td>
<td>0.1063</td>
</tr>
<tr>
<td>df1 * df2</td>
<td>21.71</td>
<td>36.63</td>
<td>0.5928</td>
</tr>
<tr>
<td>df1 + df2</td>
<td>22.04</td>
<td>36.50</td>
<td>0.6039</td>
</tr>
</tbody>
</table>

You are highly encouraged to install both libraries. See the section **Recommended Dependencies** for more installation info.

These are both enabled to be used by default, you can control this by setting the options:
2.3.4 Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`,… for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the `index` or `columns` via the `axis` keyword:

```
In [18]: df = pd.DataFrame(
    ....:    {
    ....:        "one": pd.Series(np.random.randn(3), index=["a", "b", "c"],
    ....:        "two": pd.Series(np.random.randn(4), index=["a", "b", "c", "d"],
    ....:        "three": pd.Series(np.random.randn(3), index=["b", "c", "d"],
    ....:    } )

In [19]: df
Out[19]:
     one  two  three
a  1.394981  1.772517  NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d   NaN   0.279344 -0.613172
```

```
In [20]: row = df.iloc[1]
In [21]: column = df["two"]
```

```
In [22]: df.sub(row, axis="columns")
Out[22]:
     one  two  three
a  1.051928 -0.139606  NaN
b   0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d   NaN -1.632779 -0.562782
```

```
In [23]: df.sub(row, axis=1)
Out[23]:
     one  two  three
a  1.051928 -0.139606  NaN
```
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d  NaN    -1.632779 -0.562782

In [24]: df.sub(column, axis="index")
Out[24]:
    one    two    three
   a -0.377535  0.000000    NaN
   b -1.569069  0.000000 -1.962513
   c -0.783123  0.000000 -0.250933
   d   NaN    0.000000 -0.892516

In [25]: df.sub(column, axis=0)
Out[25]:
    one    two    three
   a -0.377535  0.000000    NaN
   b -1.569069  0.000000 -1.962513
   c -0.783123  0.000000 -0.250933
   d   NaN    0.000000 -0.892516

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

In [26]: dfmi = df.copy()
In [27]: dfmi.index = pd.MultiIndex.from_tuples(
   ....: [(1, "a"), (1, "b"), (1, "c"), (2, "a")], names="first", "second"
   ....: )

In [28]: dfmi.sub(column, axis=0, level="second")
Out[28]:
    one    two    three
   first  second
     1      a -0.377535  0.000000    NaN
     2      a   NaN    -1.493173 -2.385688
     1      b -1.569069  0.000000 -1.962513
     1      c -0.783123  0.000000 -0.250933

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

In [29]: s = pd.Series(np.arange(10))
In [30]: s
Out[30]:
0  0
1  1
2  2
3  3
4  4
5  5
6  6

(continues on next page)
In [31]: div, rem = divmod(s, 3)

In [32]: div
Out[32]:
0  0
1  0
2  0
3  1
4  1
5  1
6  2
7  2
8  2
9  3
dtype: int64

In [33]: rem
Out[33]:
0  0
1  1
2  2
3  0
4  1
5  2
6  0
7  1
8  2
9  0
dtype: int64

In [34]: idx = pd.Index(np.arange(10))

In [35]: idx
Out[35]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [36]: div, rem = divmod(idx, 3)

In [37]: div
Out[37]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [38]: rem
Out[38]: Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')

We can also do elementwise divmod():

In [39]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
In [40]: div
Out[40]:
0  0  
1  0  
2  0  
3  1  
4  1  
5  1  
6  1  
7  1  
8  1  
9  1

dtype: int64

In [41]: rem
Out[41]:
0  0  
1  1  
2  2  
3  0  
4  0  
5  1  
6  1  
7  2  
8  2  
9  3

dtype: int64

Missing data / operations with fill values

In Series and DataFrame, the arithmetic functions have the option of inputting a fill_value, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using fillna if you wish).

In [42]: df
Out[42]:
     one    two    three
a  1.394981  1.772517  NaN  
b  0.343054  1.912123 -0.050390  
c  0.695246  1.478369  1.227435  
d  NaN  0.279344 -0.613172

In [43]: df2
Out[43]:
     one    two    three
a  1.394981  1.772517  1.000000  
b  0.343054  1.912123 -0.050390  
c  0.695246  1.478369  1.227435  
d  NaN  0.279344 -0.613172

2.3. Essential basic functionality
In [44]: df + df2
Out[44]:
    one  two  three
  a  2.789963  3.545034  NaN
  b   0.686107  3.824246 -0.100780
  c  1.390491  2.956737  2.454870
  d    NaN       0.558688 -1.226343

In [45]: df.add(df2, fill_value=0)
Out[45]:
    one  two  three
  a  2.789963  3.545034   1.000000
  b   0.686107  3.824246  -0.100780
  c  1.390491  2.956737   2.454870
  d    NaN       0.558688  -1.226343

Flexible comparisons

Series and DataFrame have the binary comparison methods eq, ne, lt, gt, le, and ge whose behavior is analogous to the binary arithmetic operations described above:

In [46]: df.gt(df2)
Out[46]:
    one  two  three
  a    False    False    False
  b    False    False    False
  c    False    False    False
  d    False    False    False

In [47]: df2.ne(df)
Out[47]:
    one  two  three
  a    False    False    True
  b    False    False    False
  c    False    False    False
  d     True    False    False

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype bool. These boolean objects can be used in indexing operations, see the section on Boolean indexing.

Boolean reductions

You can apply the reductions: empty, any(), all(), and bool() to provide a way to summarize a boolean result.

In [48]: (df > 0).all()
Out[48]:
    one    False
two    True
three   False
dtype: bool
You can reduce to a final boolean value.

```
In [50]: (df > 0).any().any()
Out[50]: True
```

You can test if a pandas object is empty, via the `empty` property.

```
In [51]: df.empty
Out[51]: False

In [52]: pd.DataFrame(columns=list("ABC")).empty
Out[52]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```
In [53]: pd.Series([True]).bool()
Out[53]: True

In [54]: pd.Series([False]).bool()
Out[54]: False

In [55]: pd.DataFrame([True]).bool()
Out[55]: True

In [56]: pd.DataFrame([False]).bool()
Out[56]: False
```

**Warning:** You might be tempted to do the following:

```
>>> if df:
...    pass
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.:

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See `gotchas` for a more detailed discussion.
Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider \( \text{df} + \text{df} \) and \( \text{df} \times 2 \). To test that these two computations produce the same result, given the tools shown above, you might imagine using \((\text{df} + \text{df} == \text{df} \times 2).\text{all()}\). But in fact, this expression is False:

```
In [57]: df + df == df * 2
Out[57]:
   one  two  three
a    True True  False
b    True True   True
c    True True   True
d  False True   True

In [58]: (df + df == df * 2).all()
Out[58]:
   one False
   two  True
  three False
dtype: bool
```

Notice that the boolean DataFrame \( \text{df} + \text{df} == \text{df} \times 2 \) contains some False values! This is because NaNs do not compare as equals:

```
In [59]: np.nan == np.nan
Out[59]: False
```

So, NDFrames (such as Series and DataFrames) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal:

```
In [60]: (df + df).equals(df * 2)
Out[60]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [61]: df1 = pd.DataFrame({"col": ["foo", 0, np.nan]})
In [62]: df2 = pd.DataFrame({"col": [np.nan, 0, "foo"], index=[2, 1, 0]})
In [63]: df1.equals(df2)
Out[63]: False
In [64]: df1.equals(df2.sort_index())
Out[64]: True
```
Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [65]: pd.Series(["foo", "bar", "baz"]) == "foo"
Out[65]:
0   True
1   False
2   False
dtype: bool
```

```
In [66]: pd.Index(["foo", "bar", "baz"]) == "foo"
Out[66]: array([ True, False, False])
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [67]: pd.Series(["foo", "bar", "baz"]) == pd.Index(["foo", "bar", "qux"])
Out[67]:
0   True
1   True
2   False
dtype: bool
```

```
In [68]: pd.Series(["foo", "bar", "baz"]) == np.array(["foo", "bar", "qux"])
Out[68]:
0   True
1   True
2   False
dtype: bool
```

Trying to compare Index or Series objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare
```

```
In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([2])
Out[69]: array([False, True, False])
```

or it can return False if broadcasting cannot be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```
Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of “higher quality”. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [71]: df1 = pd.DataFrame(
    ....:     {"A": [1.0, np.nan, 3.0, 5.0, np.nan], "B": [np.nan, 2.0, 3.0, np.nan, 6.0]}
    ....: )

In [72]: df2 = pd.DataFrame(
    ....:     {
    ....:         "A": [5.0, 2.0, 4.0, np.nan, 3.0, 7.0],
    ....:         "B": [np.nan, np.nan, 3.0, 4.0, 6.0, 8.0],
    ....:     }
    ....: )

In [73]: df1
Out[73]:
     A  B
0  1.0 NaN
1 NaN  2.0
2  3.0  3.0
3  5.0 NaN
4 NaN  6.0

In [74]: df2
Out[74]:
     A  B
0  5.0 NaN
1  2.0 NaN
2  4.0  3.0
3 NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [75]: df1.combine_first(df2)
Out[75]:
     A  B
0  1.0 NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```
General DataFrame combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```python
In [76]: def combiner(x, y):
    ....:     return np.where(pd.isna(x), y, x)
    ....:

In [77]: df1.combine(df2, combiner)
Out[77]:
     A  B
0  1.0 NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

2.3.5 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on Series, DataFrame. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an axis argument, just like `ndarray.{sum, std, ...}`, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: “index” (axis=0, default), “columns” (axis=1)

For example:

```python
In [78]: df
Out[78]:
     one  two  three
   a  1.394981  1.772517  NaN
   b  0.343054  1.912123 -0.050390
   c  0.695246  1.478369  1.227435
   d    NaN    0.279344 -0.613172

In [79]: df.mean(0)
Out[79]:
     one   two   three
 0  0.811094  1.360588  0.187958
```

(continues on next page)
All such methods have a `skipna` option signaling whether to exclude missing data (True by default):

```python
In [81]: df.sum(0, skipna=False)
Out[81]:
   one    two   three
b  NaN  5.442353  NaN
c  5.442353  NaN  NaN
dt不同类型: float64

In [82]: df.sum(axis=1, skipna=True)
Out[82]:
   a    b    c    d
one  3.167498  2.204786  3.401050 -0.333828
tw  5.442353  3.401050  3.401050  0.987206
tw  3.167498  2.204786  3.401050 -0.333828
dt不同类型: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation of 1), very concisely:

```python
In [83]: ts_stand = (df - df.mean()) / df.std()
In [84]: ts_stand.std()
Out[84]:
   one  two  three
one  1.0  1.0  1.0
tw  1.0  1.0  1.0
tw  1.0  1.0  1.0
dt类型: float64

In [85]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)
In [86]: xs_stand.std(1)
Out[86]:
   a  b  c  d
one  1.0  1.0  1.0  1.0
tw  1.0  1.0  1.0  1.0
tw  1.0  1.0  1.0  1.0
dt类型: float64
```

Note that methods like `cumsum()` and `cumprod()` preserve the location of NaN values. This is somewhat different from `expanding()` and `rolling()` since NaN behavior is furthermore dictated by a `min_periods` parameter.

```python
In [87]: df.cumsum()
Out[87]:
   one  two  three
a  1.394981  1.772517  NaN
b  1.738035  3.684640  -0.050390
```

(continues on next page)
Here is a quick reference summary table of common functions. Each also takes an optional `level` parameter which applies only if the object has a *hierarchical index*.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-NA observations</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>min</td>
<td>Minimum</td>
</tr>
<tr>
<td>max</td>
<td>Maximum</td>
</tr>
<tr>
<td>mode</td>
<td>Mode</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute Value</td>
</tr>
<tr>
<td>prod</td>
<td>Product of values</td>
</tr>
<tr>
<td>std</td>
<td>Bessel-corrected sample standard deviation</td>
</tr>
<tr>
<td>var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>sem</td>
<td>Standard error of the mean</td>
</tr>
<tr>
<td>skew</td>
<td>Sample skewness (3rd moment)</td>
</tr>
<tr>
<td>kurt</td>
<td>Sample kurtosis (4th moment)</td>
</tr>
<tr>
<td>quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product</td>
</tr>
<tr>
<td>cummax</td>
<td>Cumulative maximum</td>
</tr>
<tr>
<td>cummin</td>
<td>Cumulative minimum</td>
</tr>
</tbody>
</table>

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```python
In [88]: np.mean(df["one"])
Out[88]: 0.8110935116651192

In [89]: np.mean(df["one"].to_numpy())
Out[89]: nan
```

*Series.nunique()* will return the number of unique non-NA values in a Series:

```python
In [90]: series = pd.Series(np.random.randn(500))
In [91]: series[20:500] = np.nan
In [92]: series[10:20] = 5
In [93]: series.nunique()
Out[93]: 11
```
Summarizing data: describe

There is a convenient `describe()` function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```python
In [94]: series = pd.Series(np.random.randn(1000))
In [95]: series[::2] = np.nan
In [96]: series.describe()
Out[96]:
   count    500.000000
   mean    -0.021292
    std    1.015906
    min    -2.683763
   25%    -0.699070
   50%    -0.069718
   75%     0.714483
   max     3.160915
   dtype: float64
```

```python
In [97]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [98]: frame.iloc[::2] = np.nan
In [99]: frame.describe()
Out[99]:
      a          b          c          d          e
   count 500.000000  500.000000  500.000000  500.000000  500.000000
   mean  0.033387  0.030045  -0.043719  -0.051686   0.005979
    std  1.017152  0.978743  1.025270  1.015988  1.006695
   25%  -0.647623 -0.576449 -0.712369 -0.691338  -0.691115
   50%   0.047578  0.021499 -0.023888 -0.032652  -0.025363
   75%   0.729907  0.775880  0.618896  0.670047   0.649748
   max   2.740139  2.752332  3.004229  2.728702  3.240991
```

You can select specific percentiles to include in the output:

```python
In [100]: series.describe(percentiles=[0.05, 0.25, 0.75, 0.95])
Out[100]:
   count    500.000000
   mean    -0.021292
    std    1.015906
    min    -2.683763
   5%    -1.645423
   25%    -0.699070
   50%    -0.069718
   75%     0.714483
   95%     1.711409
   max     3.160915
   dtype: float64
```
By default, the median is always included.

For a non-numerical Series object, `describe()` will give a simple summary of the number of unique values and most frequently occurring values:

```
In [101]: s = pd.Series(["a", "a", "b", "b", "a", "a", np.nan, "c", "d", "a"])
In [102]: s.describe()
Out[102]:
   count   9
   unique   4
top      a
top freq  5
   dtype: object
```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [103]: frame = pd.DataFrame({'a': ["Yes", "Yes", "No", "No"], 'b': range(4)})
In [104]: frame.describe()
Out[104]:
   b
  count  4.000000
  mean   1.500000
  std    1.290994
  min    0.000000
  25%    0.750000
  50%    1.500000
  75%    2.250000
  max    3.000000
```

This behavior can be controlled by providing a list of types as `include/exclude` arguments. The special value `all` can also be used:

```
In [105]: frame.describe(include=['object'])
Out[105]:
   a
   count   4
   unique   2
top     Yes
top freq   2

In [106]: frame.describe(include=['number'])
Out[106]:
   b
  count  4.000000
  mean   1.500000
  std    1.290994
  min    0.000000
  25%    0.750000
  50%    1.500000
  75%    2.250000
  max    3.000000
```

(continues on next page)
In [107]: frame.describe(include="all")
Out[107]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4  4.000000</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>2   NaN</td>
<td></td>
</tr>
<tr>
<td>top</td>
<td>Yes  NaN</td>
<td></td>
</tr>
<tr>
<td>freq</td>
<td>2    NaN</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>NaN  1.500000</td>
<td></td>
</tr>
<tr>
<td>std</td>
<td>NaN  1.290994</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>NaN  0.000000</td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td>NaN  0.750000</td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>NaN  1.500000</td>
<td></td>
</tr>
<tr>
<td>75%</td>
<td>NaN  2.250000</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>NaN  3.000000</td>
<td></td>
</tr>
</tbody>
</table>

That feature relies on `select_dtypes`. Refer to there for details about accepted inputs.

**Index of min/max values**

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

In [108]: s1 = pd.Series(np.random.randn(5))

In [109]: s1
Out[109]:
0  1.118076
1  -0.352051
2  -1.242883
3  -1.277155
4  -0.641184
dtype: float64

In [110]: s1.idxmin(), s1.idxmax()
Out[110]: (3, 0)

In [111]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=["A", "B", "C"])

In [112]: df1
Out[112]:
   A       B       C
0  0.327863 -0.946180 -0.137570
1  0.186235 -0.257213  0.486567
2  0.507027 -0.871259 -0.111110
3  2.006339 -2.430505  0.089759
4  0.321434 -0.033695  0.096271

In [113]: df1.idxmin(axis=0)
Out[113]:
   A
0  2
When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin()` and `idxmax()` return the first matching index:

```python
In [115]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=["A"], index=list("edcba"))
In [116]: df3
Out[116]:
   A
e  2.0
d  1.0
c  1.0
b  3.0
a  NaN
In [117]: df3["A"].idxmin()
Out[117]: 'd'
```

**Note:** `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

### Value counts (histogramming) / mode

The `value_counts()` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```python
In [118]: data = np.random.randint(0, 7, size=50)
In [119]: s = pd.Series(data)
In [120]: s.value_counts()
Out[120]:
6    10
Name: A, dtype: int64
```
The `value_counts()` method can be used to count combinations across multiple columns. By default all columns are used but a subset can be selected using the `subset` argument.

```python
In [123]: data = {"a": [1, 2, 3, 4], "b": ["x", "x", "y", "y"]}
In [124]: frame = pd.DataFrame(data)
In [125]: frame.value_counts()
Out[125]:
   a b
0  x  1
1  x  1
2  y  1
3  y  1
dtype: int64
```

Similarly, you can get the most frequently occurring value(s), i.e. the mode, of the values in a Series or DataFrame:

```python
In [126]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])
In [127]: s5.mode()
Out[127]:
0  3
1  7
dtype: int64
In [128]: df5 = pd.DataFrame(
....:     {"A": np.random.randint(0, 7, size=50),
....:      "B": np.random.randint(-10, 15, size=50),
....:      })
```
In [129]: df5.mode()
Out[129]:
   A  B
0  1.0 -9
1  NaN 10
2  NaN 13

Discretization and quantiling
Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

```python
In [130]: arr = np.random.randn(20)
In [131]: factor = pd.cut(arr, 4)
In [132]: factor
Out[132]:
(-0.251, 0.464], (-0.968, -0.251], (0.464, 1.179], (-0.251, 0.464], (-0.968, -0.251], ...
Length: 20
Categories (4, interval[float64, right]): `(-0.968, -0.251] < (-0.251, 0.464] < (0.464, 1.179] < (1.79, 1.893]`

In [133]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])
In [134]: factor
Out[134]:
(0, 1], (-1, 0], (0, 1], (0, 1], (-1, 0], ..., (-1, 0], (-1, 0], (-1, 0], (-1, 0], (-1, 0])
Length: 20
Categories (4, interval[int64, right]): `(-5, -1] < (-1, 0] < (0, 1] < (1, 5]`
```

`qcut()` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```python
In [135]: arr = np.random.randn(30)
In [136]: factor = pd.qcut(arr, [0, 0.25, 0.5, 0.75, 1])
In [137]: factor
Out[137]:
(-0.301, 0.569], (-2.278, -0.301], (-2.278, -0.301], (0.569, 1.184], (0.569, 1.184], ...
Length: 30
Categories (4, interval[float64, right]): `(-2.278, -0.301] < (-0.301, 0.569] < (0.569, 1.184] < (1.184, 2.346]`
```

---

2.3. Essential basic functionality

(continues on next page)
In [138]: pd.value_counts(factor)
Out[138]:
(-2.278, -0.301]    8
(1.184, 2.346]    8
(-0.301, 0.569]   7
(0.569, 1.184]   7
dtype: int64

We can also pass infinite values to define the bins:

In [139]: arr = np.random.randn(20)

In [140]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [141]: factor
Out[141]:
[(-inf, 0.0], (0.0, inf], (0.0, inf], (-inf, 0.0], (-inf, 0.0], ..., (-inf, 0.0], (-inf,...
Length: 20
Categories (2, interval[float64, right]): [(-inf, 0.0] < (0.0, inf]]

2.3.6 Function application

To apply your own or another library’s functions to pandas objects, you should be aware of the three methods below. The
appropriate method to use depends on whether your function expects to operate on an entire DataFrame or Series,
row- or column-wise, or elementwise.

1. Tablewise Function Application: pipe()
2. Row or Column-wise Function Application: apply()
3. Aggregation API: agg() and transform()
4. Applying Elementwise Functions: applymap()

Tablewise function application

DataFrames and Series can be passed into functions. However, if the function needs to be called in a chain, consider
using the pipe() method.

First some setup:

In [142]: def extract_city_name(df):
   .....:     """
   .....:     Chicago, IL -> Chicago for city_name column
   .....:     """
   .....:     df["city_name"] = df["city_and_code"].str.split(",").str.get(0)
   .....:     return df
   .....:

In [143]: def add_country_name(df, country_name="None"):
extract_city_name and add_country_name are functions taking and returning DataFrames.

In the example above, the functions extract_city_name and add_country_name each expected a DataFrame as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide pipe with a tuple of (callable, data_keyword). pipe will route the DataFrame to the argument specified in the tuple.

For example, we can fit a regression using statsmodels. Their API expects a formula first and a DataFrame as the second argument, data. We pass in the function, keyword pair (sm.ols, 'data') to pipe:

```
In [147]: import statsmodels.formula.api as sm
In [148]: bb = pd.read_csv("data/baseball.csv", index_col="id")
```

```python
In [149]: (  
...:     bb.query("h > 0")  
...:     .assign(ln_h=lambda df: np.log(df.h))  
...:     .pipe((sm.ols, "data"), "hr ~ ln_h + year + g + C(lg)")  
...:     .fit()  
...:     .summary()  
...: )
```

```bash
Out[149]: <class 'statsmodels.iolib.summary.Summary'>
```

```python
O LS Regression Results
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

| Dep. Variable: | hr | R-squared: | 0.685 |
| Model: | OLS | Adj. R-squared: | 0.665 |
| Method: | Least Squares | F-statistic: | 34.28 |
| Date: | Sat, 02 Apr 2022 | Prob (F-statistic): | 3.48e-15 |
| Time: | 08:21:00 | Log-Likelihood: | -205.92 |
| No. Observations: | 68 | AIC: | 421.8 |
| Df Residuals: | 63 | BIC: | 432.9 |
| Df Model: | 4 |
| Covariance Type: | nonrobust |

| coef | std err | t | P>|t| | [0.025 | 0.975 |
|======|========|==|======|========|========|
| Intercept | -8484.7720 | 4664.146 | -1.819 | 0.074 | -1.78e+04 | 835.780 |
| C(lg)[T.NL] | -2.2736 | 1.325 | -1.716 | 0.091 | -4.922 | 0.375 |
| ln_h | -1.3542 | 0.875 | -1.547 | 0.127 | -3.103 | 0.395 |
| year | 4.2277 | 2.324 | 1.819 | 0.074 | -0.417 | 8.872 |
| g | 0.1841 | 0.029 | 6.258 | 0.000 | 0.125 | 0.243 |

| Omnibus: | 10.875 | Durbin-Watson: | 1.999 |
| Prob(Omnibus): | 0.004 | Jarque-Bera (JB): | 17.298 |
| Skew: | 0.537 | Prob(JB): | 0.000175 |
| Kurtosis: | 5.225 | Cond. No. | 1.49e+07 |

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are strong multicollinearity or other numerical problems.

The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular (%>%)(read pipe) operator for R. The implementation of pipe here is quite clean and feels right at home in Python. We encourage you to view the source code of pipe().

Row or column-wise function application

Arbitrary functions can be applied along the axes of a DataFrame using the `apply()` method, which, like the descriptive statistics methods, takes an optional `axis` argument:

```
In [150]: df.apply(np.mean)
Out[150]:
one    0.811094
two    1.360588
three  0.187958
dtype: float64
```

```
In [151]: df.apply(np.mean, axis=1)
Out[151]:
a     1.583749
b    0.734929
```
The `apply()` method will also dispatch on a string method name.

```
In [155]: df.apply("mean")
Out[155]:
     one   two   three
a 0.811094 1.772517     NaN
b 1.738035 3.684640 -0.050390
c 2.433281 5.163008 1.177045
d   NaN  5.442353  0.563873
```

The return type of the function passed to `apply()` affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-likes return values expand (or not) to a `DataFrame`.

---

2.3. Essential basic functionality 223
apply() combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [157]: tsdf = pd.DataFrame(
       .....:    np.random.randn(1000, 3),
       .....:    columns=['A', 'B', 'C'],
       .....:    index=pd.date_range("1/1/2000", periods=1000),
       .....:    )
       .....:
In [158]: tsdf.apply(lambda x: x.idxmax())
Out[158]:
A  2000-08-06
B  2001-01-18
C  2001-07-18
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the apply() method. For instance, consider the following function you would like to apply:

```python
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [159]: tsdf
Out[159]:
    A      B      C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
2000-01-04 NaN     NaN     NaN
2000-01-05 NaN     NaN     NaN
2000-01-06 NaN     NaN     NaN
2000-01-07 NaN     NaN     NaN
2000-01-08 -0.653602  0.178875  1.008298
2000-01-09  1.007996  0.462824  0.254472
2000-01-10  0.307473  0.600337  1.643950

In [160]: tsdf.apply(pd.Series.interpolate)
Out[160]:
    A      B      C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
2000-01-04 -1.098598 -0.889659  0.092225
2000-01-05 -0.987349 -0.622526  0.321243
2000-01-06 -0.876100 -0.355392  0.559262
2000-01-07 -0.764851 -0.088259  0.779280
2000-01-08 -0.653602  0.178875  1.008298
```

(continues on next page)
Finally, `apply()` takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

**Aggregation API**

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see `groupby API`, the `window API`, and the `resample API`. The entry point for aggregation is `DataFrame.aggregate()`, or the alias `DataFrame.agg()`.

We will use a similar starting frame from above:

```python
In [161]: tsdf = pd.DataFrame(
    .....:     np.random.randn(10, 3),
    .....:     columns=['A', 'B', 'C'],
    .....:     index=pd.date_range("1/1/2000", periods=10),
    .....:     )
```

```python
In [162]: tsdf.iloc[3:7] = np.nan
```

```python
In [163]: tsdf
Out[163]:
   A      B      C
2000-01-01  1.257606  1.004194  0.167574
2000-01-02 -0.749892  0.288112 -0.757304
2000-01-03 -0.207550 -0.298599  0.116018
2000-01-04 NaN   NaN   NaN
2000-01-05 NaN   NaN   NaN
2000-01-06 NaN   NaN   NaN
2000-01-07 NaN   NaN   NaN
2000-01-08  0.814347 -0.257623  0.869226
2000-01-09 -0.250663 -1.206601  0.896839
2000-01-10  2.169758 -1.333363  0.283157
```

Using a single function is equivalent to `apply()`. You can also pass named methods as strings. These will return a Series of the aggregated output:

```python
In [164]: tsdf.agg(np.sum)
Out[164]:
   A   B   C
2000-01-01  3.033606  1.004194  0.167574
2000-01-02  0.749892  0.288112 -0.757304
2000-01-03  0.207550 -0.298599  0.116018
2000-01-04  NaN   NaN   NaN
2000-01-05  NaN   NaN   NaN
2000-01-06  NaN   NaN   NaN
2000-01-07  NaN   NaN   NaN
2000-01-08  0.814347 -0.257623  0.869226
2000-01-09  0.250663 -1.206601  0.896839
2000-01-10  2.169758 -1.333363  0.283157
dtype: float64
```

```python
In [165]: tsdf.agg("sum")
Out[165]:
   A   B
2000-01-01  3.033606
2000-01-02 -1.803879
2000-01-03  1.575510
dtype: float64
```

(continues on next page)
C  1.575510  
dtype: float64

# these are equivalent to a `\.sum()` because we are aggregating  
# on a single function
In [166]: tsdf.sum()  
Out[166]:
A    3.033606  
B   -1.803879  
C    1.575510  
dtype: float64

Single aggregations on a Series this will return a scalar value:

In [167]: tsdf["A"].agg("sum")  
Out[167]: 3.033606102414146

Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting DataFrame. These are naturally named from the aggregation function.

In [168]: tsdf.agg(["sum"])  
Out[168]:
   A  B  C  
sum 3.033606 -1.803879 1.57551

Multiple functions yield multiple rows:

In [169]: tsdf.agg(["sum", "mean"])  
Out[169]:
   A  B  C
sum 3.033606 -1.803879 1.575510  
mean 0.505601 -0.300647 0.262585

On a Series, multiple functions return a Series, indexed by the function names:

In [170]: tsdf["A"].agg(["sum", "mean"])  
Out[170]:
   sum 3.033606  
   mean 0.505601  
Name: A, dtype: float64

Passing a lambda function will yield a <lambda> named row:

In [171]: tsdf["A"].agg(["sum", lambda x: x.mean()])  
Out[171]:
   sum 3.033606  
<lambda> 0.505601  
Name: A, dtype: float64

Passing a named function will yield that name for the row:
Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to DataFrame.agg allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an OrderedDict instead to guarantee ordering.

Passing a list-like will generate a DataFrame output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be NaN:

Mixed dtypes

Deprecated since version 1.4.0: Attempting to determine which columns cannot be aggregated and silently dropping them from the results is deprecated and will be removed in a future version. If any portion of the columns or operations provided fail, the call to .agg will raise.

When presented with mixed dtypes that cannot aggregate, .agg will only take the valid aggregations. This is similar to how .groupby.agg works.

(continues on next page)
In [177]: mdf.dtypes
Out[177]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>int64</td>
</tr>
<tr>
<td>B</td>
<td>float64</td>
</tr>
<tr>
<td>C</td>
<td>object</td>
</tr>
<tr>
<td>D</td>
<td>datetime64[ns]</td>
</tr>
</tbody>
</table>

dtype: object

In [178]: mdf.agg(['min', 'sum'])
Out[178]:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>bar</td>
<td>2013-01-01</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>NaT</td>
</tr>
</tbody>
</table>

Custom describe

With `.agg()` it is possible to easily create a custom describe function, similar to the built in `describe function`.

In [179]: from functools import partial

In [180]: q_25 = partial(pd.Series.quantile, q=0.25)

In [181]: q_25.__name__ = "25%"

In [182]: q_75 = partial(pd.Series.quantile, q=0.75)

In [183]: q_75.__name__ = "75%"

In [184]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
Out[184]:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6.000000</td>
<td>6.000000</td>
<td>6.000000</td>
</tr>
<tr>
<td>B</td>
<td>0.505601</td>
<td>-0.300647</td>
<td>0.262585</td>
</tr>
<tr>
<td>C</td>
<td>1.103362</td>
<td>0.887508</td>
<td>0.606860</td>
</tr>
<tr>
<td>D</td>
<td>-0.749892</td>
<td>-1.333363</td>
<td>-0.757304</td>
</tr>
<tr>
<td>25%</td>
<td>-0.239885</td>
<td>-0.979600</td>
<td>0.128907</td>
</tr>
<tr>
<td>median</td>
<td>0.303398</td>
<td>-0.278111</td>
<td>0.225365</td>
</tr>
<tr>
<td>75%</td>
<td>1.146791</td>
<td>0.151678</td>
<td>0.722709</td>
</tr>
<tr>
<td>max</td>
<td>2.169758</td>
<td>1.004194</td>
<td>0.896839</td>
</tr>
</tbody>
</table>

Transform API

The `transform()` method returns an object that is indexed the same (same size) as the original. This API allows you to provide multiple operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

In [185]: tsdf = pd.DataFrame(
       .....:   np.random.randn(10, 3),
       .....:   columns=['A', 'B', 'C'],
   )
Transform the entire frame. `.transform()` allows input functions as: a NumPy function, a string function name or a user defined function.

```python
In [188]: tsdf.transform(np.abs)
Out[188]:
          A         B         C
2000-01-01 0.428759  0.864890  0.675341
2000-01-02 0.168731  1.338144  1.279321
2000-01-03 1.621034  0.438107  0.903794
2000-01-04  NaN       NaN       NaN
2000-01-05  NaN       NaN       NaN
2000-01-06  NaN       NaN       NaN
2000-01-07  NaN       NaN       NaN
2000-01-08 0.254374  1.240447  0.201052
2000-01-09 0.157795  0.791197  1.144209
2000-01-10 0.030876  0.371900  0.061932

In [189]: tsdf.transform("abs")
Out[189]:
          A         B         C
2000-01-01 0.428759  0.864890  0.675341
2000-01-02 0.168731  1.338144  1.279321
2000-01-03 1.621034  0.438107  0.903794
2000-01-04  NaN       NaN       NaN
2000-01-05  NaN       NaN       NaN
2000-01-06  NaN       NaN       NaN
2000-01-07  NaN       NaN       NaN
2000-01-08 0.254374  1.240447  0.201052
2000-01-09 0.157795  0.791197  1.144209
2000-01-10 0.030876  0.371900  0.061932

In [190]: tsdf.transform(lambda x: x.abs())
(continues on next page)
Here `transform()` received a single function; this is equivalent to a `ufunc` application.

Passing a single function to `.transform()` with a Series will yield a single Series in return.
Transform with multiple functions

Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [193]: tsdf.transform([np.abs, lambda x: x + 1])
Out[193]:
     absolute  <lambda> absolute  <lambda> absolute  <lambda>
2000-01-01   0.428759  0.571241   0.864890  0.135110   0.675341  0.324659
2000-01-02   0.168731  0.831269  1.338144  2.338144  1.279321 -0.279321
2000-01-03  1.621034 -0.621034   0.438107  1.438107   0.903794  1.903794
2000-01-04   NaN      NaN      NaN      NaN      NaN      NaN
2000-01-05   NaN      NaN      NaN      NaN      NaN      NaN
2000-01-06   NaN      NaN      NaN      NaN      NaN      NaN
2000-01-07   NaN      NaN      NaN      NaN      NaN      NaN
2000-01-08  0.254374  1.254374  1.240447 -0.240447  0.201052  0.798948
2000-01-09  0.157795  0.842205  0.791197  1.791197  1.144209 -0.144209
2000-01-10  0.030876  0.969124  0.371900  1.371900  0.061932  1.061932
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [194]: tsdf["A"].transform([np.abs, lambda x: x + 1])
Out[194]:
          absolute  <lambda>
2000-01-01   0.428759  0.135110
2000-01-02   0.168731  2.338144
2000-01-03  1.621034  1.438107
2000-01-04   NaN      NaN
2000-01-05   NaN      NaN
2000-01-06   NaN      NaN
2000-01-07   NaN      NaN
2000-01-08  0.254374  1.254374
2000-01-09  0.157795  0.842205
2000-01-10  0.030876  0.969124
```

Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [195]: tsdf.transform({"A": np.abs, "B": lambda x: x + 1})
Out[195]:
          A          B
2000-01-01   0.428759  0.135110
2000-01-02   0.168731  2.338144
2000-01-03  1.621034  1.438107
2000-01-04   NaN      NaN
2000-01-05   NaN      NaN
2000-01-06   NaN      NaN
2000-01-07   NaN      NaN
2000-01-08  0.254374  0.240447
```

(continues on next page)
Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms.

In [196]: tsdf.transform({'A': np.abs, 'B': [lambda x: x + 1, 'sqrt']})

Out[196]:

    absolute  <lambda>      sqrt
2000-01-01  0.428759     0.135110     NaN
2000-01-02  0.168731     2.338144    1.156782
2000-01-03  1.621034    1.438107    0.661897
2000-01-04  NaN          NaN         NaN
2000-01-05  NaN          NaN         NaN
2000-01-06  NaN          NaN         NaN
2000-01-07  NaN          NaN         NaN
2000-01-08  0.254374    -0.240447      NaN
2000-01-09  0.157795    1.791197    0.889493
2000-01-10  0.030876    1.371900    0.609836

Applying elementwise functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods `applymap()` on DataFrame and analogously `map()` on Series accept any Python function taking a single value and returning a single value. For example:

In [197]: df4

Out[197]:

    one  two  three
a  1.394981  1.772517 NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d  NaN  0.279344  0.613172

In [198]: def f(x):
       .....: return len(str(x))
       .....:

In [199]: df4["one"].map(f)

Out[199]:

     Name: one, dtype: int64

    a  18
    b  19
    c  18
    d  3

In [200]: df4.applymap(f)

Out[200]:

    one  two  three
a  18  17  3
b  19  18  20
Series.map() has an additional feature; it can be used to easily “link” or “map” values defined by a secondary series. This is closely related to merging/joining functionality:

```python
In [201]: s = pd.Series(["six", "seven", "six", "seven", "six"], index=["a", "b", "c", "d", "e"]
   ....: )
   ....:
In [202]: t = pd.Series({"six": 6.0, "seven": 7.0})

In [203]: s
Out[203]:
a six
b seven
c six
d seven
e six
dtype: object

In [204]: s.map(t)
Out[204]:
a 6.0
b 7.0
c 6.0
d 7.0
e 6.0
dtype: float64
```

2.3.7 Reindexing and altering labels

reindex() is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To reindex means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, fill data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```python
In [205]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [206]: s
Out[206]:
a 1.695148
b 1.328614
c 1.234686
d -0.385845
dtype: float64
```
Here, the f label was not contained in the Series and hence appears as NaN in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [208]: df
Out[208]:
one   two  three
a  1.394981  1.772517  NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d   NaN      0.279344 -0.613172

In [209]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[209]:
three   two  one
  c  1.227435  1.478369  0.695246
  f   NaN      NaN      NaN
  b -0.050390  1.912123  0.343054
```

You may also use `reindex` with an `axis` keyword:

```
In [210]: df.reindex(['c', 'f', 'b'], axis='index')
Out[210]:
one   two  three
a  1.394981  1.772517  NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d   NaN      0.279344 -0.613172
```

Note that the Index objects containing the actual axis labels can be shared between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [211]: rs = s.reindex(df.index)

In [212]: rs
Out[212]:
a    1.695148
b    1.328614
c    1.234686
d -0.385845
dtype: float64

In [213]: rs.index is df.index
```

(continues on next page)
This means that the reindexed Series’s index is the same Python object as the DataFrame’s index.

`DataFrame.reindex()` also supports an “axis-style” calling convention, where you specify a single labels argument and the axis it applies to.

```
In [214]: df.reindex(["c", "f", "b"], axis="index")
Out[214]:
     one   two   three
  c  0.695246  1.478369  1.227435
  f    NaN      NaN      NaN
  b  0.343054  1.912123 -0.050390
```

```
In [215]: df.reindex(["three", "two", "one"], axis="columns")
Out[215]:
      three  two  one
  a      NaN  1.772517  1.394981
  b -0.050390  1.912123  0.343054
  c  1.227435  1.478369  0.695246
  d -0.613172  0.279344      NaN
```

See also:

`MultiIndex / Advanced Indexing` is an even more concise way of doing reindexing.

**Note:** When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: many operations are faster on pre-aligned data. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because reindex has been heavily optimized), but when CPU cycles matter sprinkling a few explicit reindex calls here and there can have an impact.

### Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like()` method is available to make this simpler:

```
In [216]: df2
Out[216]:
   one   two
  a  1.394981  1.772517
  b  0.343054  1.912123
  c  0.695246  1.478369

In [217]: df3
Out[217]:
   one   two
  a  0.583888  0.051514
  b -0.468040  0.191120
  c -0.115848 -0.242634
```
Aligning objects with each other with align

The `align()` method is the fastest way to simultaneously align two objects. It supports a join argument (related to joining and merging):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```python
In [219]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])
In [220]: s1 = s[:4]
In [221]: s2 = s[1:]
In [222]: s1.align(s2)
Out[222]:
(a -0.186646
 b -1.692424
 c -0.303893
d -1.425662
e NaN
dtype: float64,
a NaN
 b -1.692424
 c -0.303893
d -1.425662
e 1.114285
dtype: float64)
In [223]: s1.align(s2, join="inner")
Out[223]:
(b -1.692424
c -0.303893
d -1.425662
dtype: float64,
b -1.692424
c -0.303893
d -1.425662
dtype: float64)
```
For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [225]: df.align(df2, join="inner")
Out[225]:
(   one   two
   a  1.394981  1.772517
   b  0.343054  1.912123
   c  0.695246  1.478369,
    one   two
   a  1.394981  1.772517
   b  0.343054  1.912123
   c  0.695246  1.478369)
```

You can also pass an axis option to only align on the specified axis:

```
In [226]: df.align(df2, join="inner", axis=0)
Out[226]:
(   one   two   three
   a  1.394981  NaN  1.772517
   b  0.343054 -0.050390  1.912123
   c  0.695246  1.478369  1.227435,
    one   two   three
   a  1.394981  NaN  1.772517
   b  0.343054  1.912123
   c  0.695246  1.478369)
```

If you pass a Series to `DataFrame.align()`, you can choose to align both objects either on the DataFrame’s index or columns using the axis argument:

```
In [227]: df.align(df2.iloc[0], axis=1)
Out[227]:
(   one   two
   a  1.394981  NaN
   b  0.343054 -0.613172
   c  0.695246  0.279344
    one
   a  1.394981
   b  0.343054
   c  0.695246
```

(continues on next page)
Filling while reindexing

`reindex()` takes an optional parameter `method` which is a filling method chosen from the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad / ffill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
<tr>
<td>nearest</td>
<td>Fill from the nearest index value</td>
</tr>
</tbody>
</table>

We illustrate these fill methods on a simple Series:

```
In [228]: rng = pd.date_range("1/3/2000", periods=8)

In [229]: ts = pd.Series(np.random.randn(8), index=rng)

In [230]: ts2 = ts[[0, 3, 6]]

In [231]: ts
Out[231]:
          2000-01-03  0.183051
          2000-01-04  0.400528
          2000-01-05 -0.015083
          2000-01-06  2.395489
          2000-01-07  1.414806
          2000-01-08  0.118428
          2000-01-09  0.733639
          2000-01-10 -0.936077
Freq: D, dtype: float64

In [232]: ts2
Out[232]:
          2000-01-03  0.183051
          2000-01-06  2.395489
          2000-01-09  0.733639
Freq: 3D, dtype: float64

In [233]: ts2.reindex(ts.index)
Out[233]:
          2000-01-03  0.183051
          2000-01-04   NaN
          2000-01-05   NaN
          2000-01-06  2.395489
          2000-01-07   NaN
          2000-01-08   NaN
          2000-01-09  0.733639
          2000-01-10   NaN
Freq: D, dtype: float64
```
In [234]: ts2.reindex(ts.index, method="ffill")
Out[234]:
2000-01-03  0.183051
2000-01-04  0.183051
2000-01-05  0.183051
2000-01-06  2.395489
2000-01-07  2.395489
2000-01-08  2.395489
2000-01-09  0.733639
2000-01-10  0.733639
Freq: D, dtype: float64

In [235]: ts2.reindex(ts.index, method="bfill")
Out[235]:
2000-01-03  0.183051
2000-01-04  2.395489
2000-01-05  2.395489
2000-01-06  2.395489
2000-01-07  0.733639
2000-01-08  0.733639
2000-01-09  0.733639
2000-01-10  NaN
Freq: D, dtype: float64

In [236]: ts2.reindex(ts.index, method="nearest")
Out[236]:
2000-01-03  0.183051
2000-01-04  0.183051
2000-01-05  2.395489
2000-01-06  2.395489
2000-01-07  2.395489
2000-01-08  0.733639
2000-01-09  0.733639
2000-01-10  0.733639
Freq: D, dtype: float64

These methods require that the indexes are ordered increasing or decreasing.

Note that the same result could have been achieved using fillna (except for method='nearest') or interpolate:

In [237]: ts2.reindex(ts.index).fillna(method="ffill")
Out[237]:
2000-01-03  0.183051
2000-01-04  0.183051
2000-01-05  0.183051
2000-01-06  2.395489
2000-01-07  2.395489
2000-01-08  2.395489
2000-01-09  0.733639
2000-01-10  0.733639
Freq: D, dtype: float64

2.3. Essential basic functionality
reindex() will raise a ValueError if the index is not monotonically increasing or decreasing.fillna() and interpolate() will not perform any checks on the order of the index.

Limits on filling while reindexing

The limit and tolerance arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches:

```
In [238]: ts2.reindex(ts.index, method="ffill", limit=1)
Out[238]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05     NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08     NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

In contrast, tolerance specifies the maximum distance between the index and indexer values:

```
In [239]: ts2.reindex(ts.index, method="ffill", tolerance="1 day")
Out[239]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05     NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08     NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

Notice that when used on a DatetimeIndex, TimedeltaIndex or PeriodIndex, tolerance will coerced into a Timedelta if possible. This allows you to specify tolerance with appropriate strings.

Dropping labels from an axis

A method closely related to reindex is the drop() function. It removes a set of labels from an axis:

```
In [240]: df
Out[240]:
   one   two   three
  a  1.394981  1.772517   NaN
  b   0.343054  1.912123 -0.050390
  c   0.695246  1.478369  1.227435
  d     NaN   0.279344 -0.613172

In [241]: df.drop(['a', 'd'], axis=0)
Out[241]:
   one   two   three
  a  1.394981  1.772517   NaN
  b   0.343054  1.912123 -0.050390
  c   0.695246  1.478369  1.227435
  d     NaN   0.279344 -0.613172
```

(continues on next page)
Renaming / mapping labels

The `rename()` method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```python
In [244]: s
Out[244]:
   a   -0.186646
   b  -1.692424
   c  -0.303893
   d  -1.425662
   e   1.114285
dtype: float64

In [245]: s.rename(str.upper)
Out[245]:
   A   -0.186646
   B  -1.692424
   C  -0.303893
   D  -1.425662
   E   1.114285
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). A dict or Series can also be used:

```python
In [246]: df.rename(
   ....:     columns={"one": "foo", "two": "bar"},
   ....:     index={"a": "apple", "b": "banana", "d": "durian"},
   ....:     )
Out[246]:
   foo   bar   three
```

2.3. Essential basic functionality
If the mapping doesn’t include a column/index label, it isn’t renamed. Note that extra labels in the mapping don’t throw an error.

`DataFrame.rename()` also supports an “axis-style” calling convention, where you specify a single mapper and the axis to apply that mapping to.

```
In [247]: df.rename({"one": "foo", "two": "bar"}, axis="columns")
Out[247]:
   foo  bar  three
  a  1.394981  1.772517  NaN
  b  0.343054  1.912123 -0.050390
  c  0.695246  1.478369  1.227435
d    NaN      0.279344 -0.613172

In [248]: df.rename({"a": "apple", "b": "banana", "d": "durian"}, axis="index")
Out[248]:
   one   two   three
apple  1.394981  1.772517  NaN
banana  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
durian  NaN  0.279344 -0.613172
```

The `rename()` method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place.

Finally, `rename()` also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [249]: s.rename("scalar-name")
Out[249]:
   a    b    c    d
-0.186646 -1.692424 -0.303893 -1.425662 1.114285
Name: scalar-name, dtype: float64
```

The methods `DataFrame.rename_axis()` and `Series.rename_axis()` allow specific names of a MultiIndex to be changed (as opposed to the labels).

```
In [250]: df = pd.DataFrame(
       ....:     {"x": [1, 2, 3, 4, 5, 6], "y": [10, 20, 30, 40, 50, 60]},
       ....:     index=pd.MultiIndex.from_product(
       ....:         [["a", "b", "c"], [1, 2]], names=["let", "num"]
       ....:     )
       ....: )

In [251]: df
Out[251]:
```

(continues on next page)
2.3.8 Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. DataFrames follow the dict-like convention of iterating over the "keys" of the objects.

In short, basic iteration (for i in object) produces:

- **Series**: values
- **DataFrame**: column labels

Thus, for example, iterating over a DataFrame gives you the column names:

```
In [254]: df = pd.DataFrame(
       ....:     {"col1": np.random.randn(3), "col2": np.random.randn(3)},
       ....:     index=['a', 'b', 'c'])
In [255]: for col in df:
       ....:     print(col)
```

(continues on next page)
pandas objects also have the dict-like `items()` method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- **iterrows()**: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.

- **itertuples()**: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

**Warning**: Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a vectorized solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, ...

- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use `apply()` instead of iterating over the values. See the docs on function application.

- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the enhancing performance section for some examples of this approach.

**Warning**: You should never modify something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```py
In [256]: df = pd.DataFrame({"a": [1, 2, 3], "b": ["a", "b", "c"]})

In [257]: for index, row in df.iterrows():
    .....:
    row["a"] = 10
    .....:

In [258]: df
Out[258]:
    a  b
0  1  a
1  2  b
2  3  c
```
items

Consistent with the dict-like interface, `items()` iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs

For example:

```python
In [259]: for label, ser in df.items():
    ....:     print(label)
    ....:     print(ser)
    ....:
a
0  1
1  2
2  3
Name: a, dtype: int64
b
0  a
1  b
2  c
Name: b, dtype: object
```

iterrows

`iterrows()` allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```python
In [260]: for row_index, row in df.iterrows():
    ....:     print(row_index, row, sep=" \\
    ....: 
0
 a  1
b  a
Name: 0, dtype: object
1
 a  2
b  b
Name: 1, dtype: object
2
 a  3
b  c
Name: 2, dtype: object
```

Note: Because `iterrows()` returns a Series for each row, it does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```python
In [261]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
In [262]: df_orig.dtypes
Out[262]:
```

(continues on next page)
int  int64
float float64
dtype: object

In [263]: row = next(df_orig.iterrows())[1]
In [264]: row
Out[264]:
int  1.0
float 1.5
Name: 0, dtype: float64

All values in row, returned as a Series, are now upcasted to floats, also the original integer value in column x:

In [265]: row["int"]
Out[265]: dtype('float64')
In [266]: df_orig["int"]
Out[266]: dtype('int64')

To preserve dtypes while iterating over the rows, it is better to use `itertuples()` which returns namedtuples of the values and which is generally much faster than `iterrows()`.

For instance, a contrived way to transpose the DataFrame would be:

In [267]: df2 = pd.DataFrame({"x": [1, 2, 3], "y": [4, 5, 6]})
In [268]: print(df2)
x y
0 1 4
1 2 5
2 3 6
In [269]: print(df2.T)
   0 1 2
   x 1 2 3
   y 4 5 6
In [270]: df2_t = pd.DataFrame({idx: values for idx, values in df2.iterrows()})
In [271]: print(df2_t)
   0 1 2
   x 1 2 3
   y 4 5 6
**itertuples**

The `itertuples()` method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the row’s corresponding index value, while the remaining values are the row values. For instance:

```python
In [272]: for row in df.itertuples():
    ...:     print(row)
    ...:
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, `itertuples()` preserves the data type of the values and is generally faster as `iterrows()`.

**Note:** The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

### 2.3.9 .dt accessor

Series has an accessor to succinctly return datetime like properties for the `values` of the Series, if it is a datetime/period like Series. This will return a Series, indexed like the existing Series.

```python
# datetime
In [273]: s = pd.Series(pd.date_range("20130101 09:10:12", periods=4))

In [274]: s
Out[274]:
0  2013-01-01 09:10:12
1  2013-01-02 09:10:12
2  2013-01-03 09:10:12
3  2013-01-04 09:10:12
dtype: datetime64[ns]

In [275]: s.dt.hour
Out[275]:
0    9
1    9
2    9
3    9
dtype: int64

In [276]: s.dt.second
Out[276]:
0   12
1   12
2   12
3   12
dtype: int64
```

(continues on next page)
This enables nice expressions like this:

```
In [278]: s[s.dt.day == 2]
Out[278]:
1  2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [279]: stz = s.dt.tz_localize("US/Eastern")
In [280]: stz
Out[280]:
0  2013-01-01 09:10:12-05:00
1  2013-01-02 09:10:12-05:00
2  2013-01-03 09:10:12-05:00
3  2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also chain these types of operations:

```
In [282]: s.dt.tz_localize("UTC").dt.tz_convert("US/Eastern")
Out[282]:
0  2013-01-01 04:10:12-05:00
1  2013-01-02 04:10:12-05:00
2  2013-01-03 04:10:12-05:00
3  2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also format datetime values as strings with `Series.dt.strftime()` which supports the same format as the standard `strftime()`.

```
In [283]: s = pd.Series(pd.date_range("20130101", periods=4))
In [284]: s
Out[284]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: datetime64[ns]
```

# DatetimeIndex

```
In [285]: s = pd.Series(pd.date_range("20130101", periods=4))
In [286]: s
Out[286]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: datetime64[ns]
```
In [285]: s.dt.strftime("%Y/%m/%d")
Out[285]:
0  2013/01/01
1  2013/01/02
2  2013/01/03
3  2013/01/04
dtype: object

# PeriodIndex
In [286]: s = pd.Series(pd.period_range("20130101", periods=4))

In [287]: s
Out[287]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: period[D]

In [288]: s.dt.strftime("%Y/%m/%d")
Out[288]:
0  2013/01/01
1  2013/01/02
2  2013/01/03
3  2013/01/04
dtype: object

The .dt accessor works for period and timedelta dtypes.

# period
In [289]: s = pd.Series(pd.period_range("20130101", periods=4, freq="D"))

In [290]: s
Out[290]:
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
dtype: period[D]

In [291]: s.dt.year
Out[291]:
0  2013
1  2013
2  2013
3  2013
dtype: int64

In [292]: s.dt.day
Out[292]:
0  1
# timedelta

In [293]: s = pd.Series(pd.timedelta_range("1 day 00:00:05", periods=4, freq="s"))

In [294]: s
Out[294]:
0  1 days 00:00:05
1  1 days 00:00:06
2  1 days 00:00:07
3  1 days 00:00:08
dtype: timedelta64[ns]

In [295]: s.dt.days
Out[295]:
0  1
1  1
2  1
3  1
dtype: int64

In [296]: s.dt.seconds
Out[296]:
0  5
1  6
2  7
3  8
dtype: int64

In [297]: s.dt.components
Out[297]:
   days  hours  minutes  seconds milliseconds microseconds nanoseconds
0   1      0        0       5          0          0           0
1   1      0        0       6          0          0           0
2   1      0        0       7          0          0           0
3   1      0        0       8          0          0           0

Note: Series.dt will raise a TypeError if you access with a non-datetime-like values.
2.3.10 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series’s `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```
In [298]: s = pd.Series(
    .....:     dtype="string"
    .....: )

In [299]: s.str.lower()
Out[299]:
0   a
1   b
2   c
3  aaba
4  baca
5  <NA>
6   caba
7    dog
8    cat
dtype: string
```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them).

**Note:** Prior to pandas 1.0, string methods were only available on `object`-dtype `Series`. pandas 1.0 added the `StringDtype` which is dedicated to strings. See Text data types for more.

Please see Vectorized String Methods for a complete description.

2.3.11 Sorting

pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

**By index**

The `Series.sort_index()` and `DataFrame.sort_index()` methods are used to sort a pandas object by its index levels.

```
In [300]: df = pd.DataFrame(
    .....:     {
    .....:         "one": pd.Series(np.random.randn(3), index=["a", "b", "c"]),
    .....:         "two": pd.Series(np.random.randn(4), index=["a", "b", "c", "d"]),
    .....:         "three": pd.Series(np.random.randn(3), index=["b", "c", "d"]),
    .....:     }
    .....: )
```

(continues on next page)
In [301]: unsorted_df = df.reindex(
       .....:   index=["a", "d", "c", "b"], columns=["three", "two", "one"]
       .....: )
       .....:
In [302]: unsorted_df
Out[302]:
   three  two  one
  a    NaN  -1.152244  0.562973
  d  -0.252916  -0.109597    NaN
  c  1.273388  -0.167123  0.640382
  b  -0.098217   0.009797 -1.299504

# DataFrame
In [303]: unsorted_df.sort_index()
Out[303]:
   three  two  one
  a    NaN  -1.152244  0.562973
  b  -0.098217   0.009797 -1.299504
  c  1.273388  -0.167123  0.640382
  d  -0.252916  -0.109597    NaN

In [304]: unsorted_df.sort_index(ascending=False)
Out[304]:
   three  two  one
  d  -0.252916  -0.109597    NaN
  c  1.273388  -0.167123  0.640382
  b  -0.098217   0.009797 -1.299504
  a    NaN  -1.152244  0.562973

In [305]: unsorted_df.sort_index(axis=1)
Out[305]:
   one  three  two
  a  0.562973    NaN  -1.152244
  d    NaN  -0.252916  -0.109597
  c  0.640382  1.273388  -0.167123
  b -1.299504  -0.098217   0.009797

# Series
In [306]: unsorted_df["three"].sort_index()
Out[306]:
   Name: three, dtype: float64

New in version 1.1.0.

Sorting by index also supports a key parameter that takes a callable function to apply to the index being sorted. For MultiIndex objects, the key is applied per-level to the levels specified by level.
In [307]: s1 = pd.DataFrame({"a": ["B", "a", "C"], "b": [1, 2, 3], "c": [2, 3, 4]}).set_index(list("ab"))

In [308]: s1
Out[308]:
  c
  a  b
  B 1 2
  a 2 3
  C 3 4

In [309]: s1.sort_index(level="a")
Out[309]:
  c
  a  b
  B 1 2
  a 2 3
  C 3 4

In [310]: s1.sort_index(level="a", key=lambda idx: idx.str.lower())
Out[310]:
  c
  a  b
  a 2 3
  B 1 2
  C 3 4

For information on key sorting by value, see value sorting.

By values

The Series.sort_values() method is used to sort a Series by its values. The DataFrame.sort_values() method is used to sort a DataFrame by its column or row values. The optional by parameter to DataFrame.sort_values() may be used to specify one or more columns to use to determine the sorted order.

In [311]: df1 = pd.DataFrame(
               ......:     {"one": [2, 1, 1, 1], "two": [1, 3, 2, 4], "three": [5, 4, 3, 2]}
               ......: )
               ......:

In [312]: df1.sort_values(by="two")
Out[312]:
  one  two  three
  0  2  1  5
  2  1  2  3
  1  1  3  4
  3  1  4  2

The by parameter can take a list of column names, e.g.:
These methods have special treatment of NA values via the `na_position` argument:

```python
In [314]: s[2] = np.nan

In [315]: s.sort_values()
Out[315]:
   0  A
   3  Aaba
   1  B
   4  Baca
   6  CABA
   8  cat
   7  dog
   2  <NA>
   5  <NA>

   dtype: string

In [316]: s.sort_values(na_position="first")
Out[316]:
   2  <NA>
   5  <NA>
   0  A
   3  Aaba
   1  B
   4  Baca
   6  CABA
   8  cat
   7  dog

   dtype: string
```

New in version 1.1.0.

Sorting also supports a `key` parameter that takes a callable function to apply to the values being sorted.

```python
In [317]: s1 = pd.Series(['B', 'a', 'C'])

In [318]: s1.sort_values()
Out[318]:
   0  B
   2  C
   1  a

   dtype: object

In [319]: s1.sort_values(key=lambda x: x.str.lower())
Out[319]:
```

(continues on next page)
key will be given the Series of values and should return a Series or array of the same shape with the transformed values. For DataFrame objects, the key is applied per column, so the key should still expect a Series and return a Series, e.g.

```
In [320]: df = pd.DataFrame(
    ...:     {"a": ["B", "a", "C"], "b": [1, 2, 3]})
```

```
In [321]: df.sort_values(by="a")
Out[321]:
   a   b
0  B   1
2  C   3
1  a   2
```

```
In [322]: df.sort_values(by="a", key=lambda col: col.str.lower())
```

The name or type of each column can be used to apply different functions to different columns.

**By indexes and values**

Strings passed as the by parameter to DataFrame.sort_values() may refer to either columns or index level names.

```
# Build MultiIndex
In [323]: idx = pd.MultiIndex.from_tuples((
    ...:     [("a", 1), ("a", 2), ("a", 2), ("b", 2), ("b", 1), ("b", 1)]
    ...:     ))
```

```
In [324]: idx.names = ["first", "second"]
```

```
# Build DataFrame
In [325]: df_multi = pd.DataFrame({"A": np.arange(6, 0, -1)}, index=idx)
```

```
In [326]: df_multi
Out[326]:
     A  first  second
   a
  1  6
  2  5
  2  4
   b
  2  3
  1  2
  1  1
```

2.3. Essential basic functionality 255
Sort by ‘second’ (index) and ‘A’ (column)

```
In [327]: df_multi.sort_values(by=['second', 'A'])
Out[327]:
   A
first second
b   1   1
    1   2
a   1   6
b   2   3
a   2   4
    2   5
```

**Note:** If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

**searchsorted**

Series has the `searchsorted()` method, which works similarly to `numpy.ndarray.searchsorted()`.

```
In [328]: ser = pd.Series([1, 2, 3])
In [329]: ser.searchsorted([0, 3])
Out[329]: array([0, 2])
In [330]: ser.searchsorted([0, 4])
Out[330]: array([0, 3])
In [331]: ser.searchsorted([1, 3], side="right")
Out[331]: array([1, 3])
In [332]: ser.searchsorted([1, 3], side="left")
Out[332]: array([0, 2])
In [333]: ser = pd.Series([3, 1, 2])
In [334]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[334]: array([0, 2])
```

**smallest / largest values**

Series has the `nsmallest()` and `nlargest()` methods which return the smallest or largest $n$ values. For a large Series this can be much faster than sorting the entire Series and calling `head(n)` on the result.

```
In [335]: s = pd.Series(np.random.permutation(10))
In [336]: s
Out[336]:
   0   2
   1   0
   ... 
```

(continues on next page)
2.3. Essential basic functionality

In [337]: s.sort_values()
Out[337]:
   1  0
   4  1
   0  2
   2  3
   9  4
   5  5
   7  6
   3  7
   8  8
   6  9
dtype: int64

In [338]: s.nsmallest(3)
Out[338]:
   1  0
   4  1
   0  2
dtype: int64

In [339]: s.nlargest(3)
Out[339]:
   6  9
   8  8
   3  7
dtype: int64

DataFrame also has the nlargest and nsmallest methods.

In [340]: df = pd.DataFrame(
   ....:     { 
   ....:         "a": [-2, -1, 1, 10, 8, 11, -1],
   ....:         "b": list("abdceff"),
   ....:         "c": [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0],
   ....:     }
   ....: )
   ....:

In [341]: df.nlargest(3, "a")
Out[341]:
   a  b  c
   10 2 4.0
   11 2 4.0
   11 2 4.0

(continues on next page)
In [342]: df.nlargest(5, ["a", "c"])
Out[342]:
   a  b  c
5 11  f  3.0
3 10  c  3.2
4  8  e  NaN

In [343]: df.nsmallest(3, "a")
Out[343]:
   a  b  c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0

In [344]: df.nsmallest(5, ["a", "c"])
Out[344]:
   a  b  c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0
2  1  d  4.0
4  8  e  NaN

Sorting by a MultiIndex column

You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to by.

In [345]: df1.columns = pd.MultiIndex.from_tuples(
       ......: ["a", "one"), ("a", "two"), ("b", "three")]
       ......: )

In [346]: df1.sort_values(by="a", "two")
Out[346]:
   a  b
one two three
  0 2 1  5
  2 1 2  3
  1 1 3  4
  3 1 4  2
### 2.3.12 Copying

The `copy()` method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column.
- Assigning to the index or columns attributes.
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing.

To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

### 2.3.13 dtypes

For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for `float, int, bool, timedelta64[ns]` and `datetime64[ns]` (note that NumPy does not support timezone-aware datetimes).

Pandas and third-party libraries extend NumPy’s type system in a few places. This section describes the extensions pandas has made internally. See [*Extension types*](#) for how to write your own extension that works with pandas. See [ecosystem.extensions](#) for a list of third-party libraries that have implemented an extension.

The following table lists all of pandas extension types. For methods requiring `dtype` arguments, strings can be specified as indicated. See the respective documentation sections for more on each type.

<table>
<thead>
<tr>
<th>Kind of Data</th>
<th>Data Type</th>
<th>Scalar</th>
<th>Array</th>
<th>String Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>tz-aware date-time</td>
<td><code>DatetimeTZDtype</code></td>
<td><code>Timestamp</code></td>
<td><code>arrays. DatetimeArray</code></td>
<td><code>'datetime64[ns, &lt;tz&gt;]'</code></td>
</tr>
<tr>
<td>Categorical</td>
<td><code>CategoricalDtype</code></td>
<td><code>category</code></td>
<td><code>Categorical</code></td>
<td><code>'category'</code></td>
</tr>
<tr>
<td>period (time spans)</td>
<td><code>PeriodDtype</code></td>
<td><code>Period</code></td>
<td><code>arrays. PeriodArray</code></td>
<td><code>'period[&lt;freq&gt;]',</code></td>
</tr>
<tr>
<td>sparse</td>
<td><code>SparseDtype</code></td>
<td>(none)</td>
<td><code>arrays. SparseArray</code></td>
<td><code>'Sparse', 'Sparse[int]', 'Sparse[float]'</code></td>
</tr>
<tr>
<td>intervals</td>
<td><code>IntervalDtype</code></td>
<td><code>Interval</code></td>
<td><code>arrays. IntervalArray</code></td>
<td><code>'Interval', </code></td>
</tr>
</tbody>
</table>
| nullable integer | `Int64Dtype` | (none) | `arrays. IntegerArray` | `'Int8', 'Int16', 'Int32', 'Int64', 'UInt8',
 'UInt16', 'UInt32', 'UInt64' |
| Strings | `StringDtype` | `str` | `arrays. StringArray` | `'string'` |
| Boolean (with NA) | `BooleanDtype` | `bool` | `arrays. BooleanArray` | `'boolean'` |
pandas has two ways to store strings.

1. object dtype, which can hold any Python object, including strings.

2. StringDtype, which is dedicated to strings.

Generally, we recommend using StringDtype. See Text data types for more.

Finally, arbitrary objects may be stored using the object dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See object conversion).

A convenient dtypes attribute for DataFrame returns a Series with the data type of each column.

```python
In [347]: dft = pd.DataFrame(
    ..:    {  
    ..:       "A": np.random.rand(3),  
    ..:       "B": 1,  
    ..:       "C": "foo",  
    ..:       "D": pd.Timestamp("20010102"),  
    ..:       "E": pd.Series([1.0] * 3).astype("float32"),  
    ..:       "F": False,  
    ..:       "G": pd.Series([1] * 3, dtype="int8"),  
    ..:    }  
    ..: )

In [348]: dft
Out[348]:
     A    B   C       D       E       F   G
0  0.035962  1.0  foo 2001-01-02  1.0 False  1
1  0.701379  1.0  foo 2001-01-02  1.0 False  1
2  0.281885  1.0  foo 2001-01-02  1.0 False  1

In [349]: dft.dtypes
Out[349]:
A     float64
B      int64
C     object
D   datetime64[ns]
E     float32
F      bool
G     int8
dtype: object
```

On a Series object, use the dtype attribute.

```python
In [350]: dft["A"].dtype
Out[350]: dtype('float64')
```

If a pandas object contains data with multiple dtypes in a single column, the dtype of the column will be chosen to accommodate all of the data types (object is the most general).

```python
# these ints are coerced to floats
In [351]: pd.Series([1, 2, 3, 4, 5, 6.0])
Out[351]:
0    1.0
```

(continues on next page)
# string data forces an `object` dtype

```python
In [352]: pd.Series([1, 2, 3, 6.0, "foo")
```

```
Out[352]:
0 1
1 2
2 3
3 6.0
4 foo
dtype: object
```

The number of columns of each type in a DataFrame can be found by calling `DataFrame.dtypes.value_counts()`.

```python
In [353]: dft.dtypes.value_counts()
```

```
Out[353]:
float64 1
int64 1
object 1
datetime64[ns] 1
float32 1
bool 1
int8 1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```python
In [354]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=["A"], dtype="float32")
```

```python
In [355]: df1
```

```
Out[355]:
     A
0 0.224364
1 1.890546
2 0.182879
3 0.787847
4 -0.188449
5 0.667715
6 -0.011736
7 -0.399073
```

```python
In [356]: df1.dtypes
```

```
Out[356]:
A    float32
dtype: object
```

(continues on next page)
In [357]: df2 = pd.DataFrame(
    ......:     {'A': pd.Series(np.random.randn(8), dtype='float16'),
    ......:     'B': pd.Series(np.random.randn(8)),
    ......:     'C': pd.Series(np.array(np.random.randn(8), dtype='uint8'))},
    ......: )

In [358]: df2
Out[358]:
       A       B      C
0  0.823242  0.256090  0
1  1.607422  1.426469  0
2 -0.333740 -0.416203 255
3 -0.063477  1.139976  0
4 -1.014648 -1.193477  0
5  0.678711  0.096706  0
6 -0.040863 -1.956850  1
7 -0.357422 -0.714337  0

In [359]: df2.dtypes
Out[359]:
A  float16
B  float64
C  uint8
dtype: object

defaults

By default integer types are int64 and float types are float64, regardless of platform (32-bit or 64-bit). The following will all result in int64 dtypes.

In [360]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[360]:
a  int64
dtype: object

In [361]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[361]:
a  int64
dtype: object

In [362]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[362]:
a  int64
dtype: object

Note that Numpy will choose platform-dependent types when creating arrays. The following WILL result in int32 on 32-bit platform.
upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [364]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```

```
In [365]: df3
Out[365]:
      A       B       C
0  1.04761  0.25609  0.0
1  3.49797  1.42647  0.0
2 -0.15086 -0.41620  255.0
3  0.72437  1.13998  0.0
4 -1.20309 -1.19348  0.0
5  1.34643  0.09670  0.0
6 -0.05259 -0.71434  0.0
7 -0.75649 -0.71434  0.0
```

```
In [366]: df3.dtypes
Out[366]:
A    float32
B    float64
C    float64
dtype: object
```

`DataFrame.to_numpy()` will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate ALL of the types in the resulting homogeneous dtype NumPy array. This can force some *upcasting*.

```
In [367]: df3.to_numpy().dtype
Out[367]: dtype('float64')
```

*astype*

You can use the `astype()` method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the NumPy rules. If two different dtypes are involved in an operation, then the more general one will be used as the result of the operation.

```
In [368]: df3
Out[368]:
      A       B       C
0  1.04761  0.25609  0.0
1  3.49797  1.42647  0.0
2 -0.15086 -0.41620  255.0
3  0.72437  1.13998  0.0
4 -1.20309 -1.19348  0.0
```
In [369]: df3.dtypes
Out[369]:
A float32
B float64
C float64
dtype: object

# conversion of dtypes
In [370]: df3.astype("float32").dtypes
Out[370]:
A float32
B float32
C float32
dtype: object

Convert a subset of columns to a specified type using *astype()*.

In [371]: dft = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})

In [372]: dft["a", "b"] = dft["a", "b"].astype(np.uint8)

In [373]: dft
Out[373]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [374]: dft.dtypes
Out[374]:
a  uint8
b  uint8
c  int64
dtype: object

Convert certain columns to a specific dtype by passing a dict to *astype()*.

In [375]: dft1 = pd.DataFrame({"a": [1, 0, 1], "b": [4, 5, 6], "c": [7, 8, 9]})

In [376]: dft1 = dft1.astype({"a": np.bool_, "c": np.float64})

In [377]: dft1
Out[377]:
   a    b    c
0  True  4.0
1  False  5.0
2  True  6.0
Note: When trying to convert a subset of columns to a specified type using `astype()` and `loc()`, upcasting occurs. `loc()` tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```python
In [379]: dft = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})
In [380]: dft.loc[:, ["a", "b"]].astype(np.uint8).dtypes
Out[380]:
   a uint8
   b uint8
dtype: object

In [381]: dft.loc[:, ["a", "b"]].astype(np.uint8)
In [382]: dft.dtypes
Out[382]:
   a int64
   b int64
c int64
dtype: object
```

**object conversion**

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the `DataFrame.infer_objects()` and `Series.infer_objects()` methods can be used to soft convert to the correct type.

```python
In [383]: import datetime

In [384]: df = pd.DataFrame(
        .....:             [1, 2],
        .....:             ["a", "b"],
        .....:             [datetime.datetime(2016, 3, 2),
        .....:             datetime.datetime(2016, 3, 2)],
    ....)
In [385]: df = df.T
In [386]: df
```

(continues on next page)
Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- `to_numeric()` (conversion to numeric dtypes)

```python
In [390]: m = ["1.1", 2, 3]
In [390]: pd.to_numeric(m)
Out[390]: array([1.1, 2., 3.])
```

- `to_datetime()` (conversion to datetime objects)

```python
In [391]: import datetime
In [392]: m = ["2016-07-09", datetime.datetime(2016, 3, 2)]
In [393]: pd.to_datetime(m)
Out[393]: DatetimeIndex(["2016-07-09", '2016-03-02'], dtype='datetime64[ns]', freq=None)
```

- `to_timedelta()` (conversion to timedelta objects)

```python
In [394]: m = ["5us", pd.Timedelta("1day")]
In [395]: pd.to_timedelta(m)
Out[395]: TimedeltaIndex(["0 days 00:00:00.000005", '1 days 00:00:00'], dtype='timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be
useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```python
In [396]: import datetime
In [397]: m = ['apple', datetime.datetime(2016, 3, 2)]
In [398]: pd.to_datetime(m, errors='coerce')
Out[398]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)
In [399]: m = ['apple', 2, 3]
In [400]: pd.to_numeric(m, errors='coerce')
Out[400]: array(['apple', 2., 3.], dtype=object)
In [401]: m = ['apple', pd.Timedelta('1day')]
In [402]: pd.to_timedelta(m, errors='coerce')
Out[402]: TimedeltaIndex(['NaT', '1 days'], dtype='timedelta64[ns]', freq=None)
```

The errors parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```python
In [403]: import datetime
In [404]: m = ['apple', datetime.datetime(2016, 3, 2)]
In [405]: pd.to_datetime(m, errors='ignore')
Out[405]: Index(['apple', 2016-03-02 00:00:00], dtype=object)
In [406]: m = ['apple', 2, 3]
In [407]: pd.to_numeric(m, errors='ignore')
Out[407]: array(['apple', 2, 3], dtype=object)
In [408]: m = ['apple', pd.Timedelta('1day')]
In [409]: pd.to_timedelta(m, errors='ignore')
Out[409]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, `to_numeric()` provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```python
In [410]: m = ['1', 2, 3]
In [411]: pd.to_numeric(m, downcast='integer')  # smallest signed int dtype
Out[411]: array([1, 2, 3], dtype=int8)
In [412]: pd.to_numeric(m, downcast='signed')  # same as 'integer'
Out[412]: array([1, 2, 3], dtype=int8)
In [413]: pd.to_numeric(m, downcast='unsigned')  # smallest unsigned int dtype
Out[413]: array([1, 2, 3], dtype=uint8)
```
As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with `apply()`, we can “apply” the function over each column efficiently:

```
In [415]: import datetime

In [416]: df = pd.DataFrame([["2016-07-09", datetime.datetime(2016, 3, 2)]] * 2, dtype="O"

In [417]: df
Out[417]:
   0         1
0 2016-07-09 2016-03-02 00:00:00
1 2016-07-09 2016-03-02 00:00:00

In [418]: df.apply(pd.to_datetime)
Out[418]:
   0         1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [419]: df = pd.DataFrame(["1.1", 2, 3] * 2, dtype="O")

In [420]: df
Out[420]:
   0 1 2
0 1.1 2 3
1 1.1 2 3

In [421]: df.apply(pd.to_numeric)
Out[421]:
   0 1 2
0 1.1 2 3
1 1.1 2 3

In [422]: df = pd.DataFrame(["5us", pd.Timedelta("1day")]] * 2, dtype="O")

In [423]: df
Out[423]:
   0         1
0 5us 1 days 00:00:00
1 5us 1 days 00:00:00

In [424]: df.apply(pd.to_timedelta)
Out[424]:
   0         1
0 0 days 00:00:00.000005 1 days
1 0 days 00:00:00.000005 1 days
```
gotchas

Performing selection operations on integer type data can easily upcast the data to floating. The dtype of the input data will be preserved in cases where nans are not introduced. See also Support for integer NA.

```python
In [425]: dfi = df3.astype("int32")
In [426]: dfi["E"] = 1
In [427]: dfi
Out[427]:
   A  B  C  E
0  0  1  0  1
1  1  3  1  1
2  0  0  255 1
3  0  1  0  1
4 -1 -1  0  1
5  1  0  0  1
6  0 -1  1  1
7  0  0  0  1
```

```python
In [428]: dfi.dtypes
Out[428]:
A   int32
B   int32
C   int32
E   int64
dtype: object
```

```python
In [429]: casted = dfi[dfi > 0]
In [430]: casted
Out[430]:
   A  B  C  E
0  1.0 NaN NaN  1
1  3.0 1.0  NaN  1
2 NaN  NaN  255.0  1
3 NaN  NaN  1.0  NaN
4 NaN  NaN  NaN  1
5  1.0  NaN  NaN  1
6 NaN  NaN  1.0  1
7 NaN  NaN  NaN  1
```

```python
In [431]: casted.dtypes
Out[431]:
A   float64
B   float64
C   float64
E   int64
dtype: object
```

While float dtypes are unchanged.
2.3.14 Selecting columns based on dtype

The `select_dtypes()` method implements subsetting of columns based on their dtype.

First, let’s create a DataFrame with a slew of different dtypes:

```
In [438]: df = pd.DataFrame(
    ....:     {  
    ....:         "string": list("abc"),  
    ....:         "int64": list(range(1, 4)),  
    ....:         "uint8": np.arange(3, 6).astype("u1"),  
    ....:         "float64": np.arange(4.0, 7.0),  
    ....:         "bool1": [True, False, True],  
    ....:         "bool2": [False, True, False],  
    ....:         "dates": pd.date_range("now", periods=3),  
    ....:         "category": pd.Series(list("ABC")).astype("category"),  
    ....:     }
    ....: )
```

(continues on next page)
In [439]: df["tdeltas"] = df.dates.diff()

In [440]: df["uint64"] = np.arange(3, 6).astype("u8")

In [441]: df["other_dates"] = pd.date_range("20130101", periods=3)

In [442]: df["tz_aware_dates"] = pd.date_range("20130101", periods=3, tz="US/Eastern")

In [443]: df

<table>
<thead>
<tr>
<th>string</th>
<th>int64</th>
<th>uint8</th>
<th>float64</th>
<th>bool1</th>
<th>...</th>
<th>category</th>
<th>tdeltas</th>
<th>uint64</th>
<th>other_dates</th>
<th>tz_aware_dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>3</td>
<td>4.0</td>
<td>True</td>
<td>...</td>
<td>A</td>
<td>NaT</td>
<td>3</td>
<td>2013-01-01</td>
<td>2013-01-00:00:00-05:00</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
<td>5.0</td>
<td>False</td>
<td>...</td>
<td>B</td>
<td>1 days</td>
<td>4</td>
<td>2013-01-02</td>
<td>2013-01-00:00:00-05:00</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>5</td>
<td>6.0</td>
<td>True</td>
<td>...</td>
<td>C</td>
<td>1 days</td>
<td>5</td>
<td>2013-01-03</td>
<td>2013-01-00:00:00-05:00</td>
</tr>
</tbody>
</table>

[3 rows x 12 columns]

And the dtypes:

In [444]: df.dtypes

<table>
<thead>
<tr>
<th>string</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64</td>
<td>int64</td>
</tr>
<tr>
<td>uint8</td>
<td>uint8</td>
</tr>
<tr>
<td>float64</td>
<td>float64</td>
</tr>
<tr>
<td>bool1</td>
<td>bool</td>
</tr>
<tr>
<td>bool2</td>
<td>bool</td>
</tr>
<tr>
<td>dates</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>category</td>
<td>category</td>
</tr>
<tr>
<td>tdeltas</td>
<td>timedelta64[ns]</td>
</tr>
<tr>
<td>uint64</td>
<td>uint64</td>
</tr>
<tr>
<td>other_dates</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>tz_aware_dates</td>
<td>datetime64[ns, US/Eastern]</td>
</tr>
</tbody>
</table>

dtype: object

`select_dtypes()` has two parameters `include` and `exclude` that allow you to say “give me the columns with these dtypes” (include) and/or “give the columns without these dtypes” (exclude).

For example, to select bool columns:

In [445]: df.select_dtypes(include=[bool])

Out[445]:

<table>
<thead>
<tr>
<th>bool1</th>
<th>bool2</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

You can also pass the name of a dtype in the NumPy dtype hierarchy:

2.3. Essential basic functionality
In [446]: df.select_dtypes(include=['bool'])
Out[446]:
   bool1  bool2
0   True  False
1  False   True
2   True  False

select_dtypes() also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

In [447]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[447]:
   int64    float64  bool1  bool2  tdeltas
0  1.00    4.000000   True  False  NaT
1  2.00    5.000000  False   True   1 days
2  3.00    6.000000   True  False   1 days

To select string columns you must use the object dtype:

In [448]: df.select_dtypes(include=['object'])
Out[448]:
   string
0    a
1    b
2    c

To see all the child dtypes of a generic dtype like numpy.number you can define a function that returns a tree of child dtypes:

In [449]: def subdtypes(dtype):
    ...
    subs = dtype.__subclasses__()
    if not subs:
        return dtype
    return [dtype, [subdtypes(dt) for dt in subs]]

All NumPy dtypes are subclasses of numpy.generic:

In [450]: subtypes(np.generic)
Out[450]:
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,  
numpy.int16,  
numpy.int32,  
numpy.int64,  
numpy.longlong,  
numpy.timedelta64]],
       [numpy.unsignedinteger,
         [numpy.uint8,  
numpy.uint16,  
numpy.uint32,  
numpy.uint64,  
numpy.ulonglong,  
numpy.timedelta64]],
       [numpy.complex]  
numpy.complex64,  
numpy.complex128]]]]]
2.4 IO tools (text, CSV, HDF5, …)

The pandas I/O API is a set of top level reader functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding writer functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available readers and writers.

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Data Description</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>CSV</td>
<td><code>read_csv</code></td>
<td><code>to_csv</code></td>
</tr>
<tr>
<td>text</td>
<td>Fixed-Width Text File</td>
<td><code>read_fwf</code></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>JSON</td>
<td><code>read_json</code></td>
<td><code>to_json</code></td>
</tr>
<tr>
<td>text</td>
<td>HTML</td>
<td><code>read_html</code></td>
<td><code>to_html</code></td>
</tr>
<tr>
<td>text</td>
<td>LaTeX</td>
<td><code>to_latex</code></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>XML</td>
<td><code>read_xml</code></td>
<td><code>to_xml</code></td>
</tr>
<tr>
<td>text</td>
<td>Local clipboard</td>
<td><code>read_clipboard</code></td>
<td><code>to_clipboard</code></td>
</tr>
<tr>
<td>binary</td>
<td>MS Excel</td>
<td><code>read_excel</code></td>
<td><code>to_excel</code></td>
</tr>
<tr>
<td>binary</td>
<td>OpenDocument</td>
<td><code>read_excel</code></td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>HDF5 Format</td>
<td><code>read_hdf</code></td>
<td><code>to_hdf</code></td>
</tr>
<tr>
<td>binary</td>
<td>Feather Format</td>
<td><code>read_feather</code></td>
<td><code>to_feather</code></td>
</tr>
<tr>
<td>binary</td>
<td>Parquet Format</td>
<td><code>read_parquet</code></td>
<td><code>to_parquet</code></td>
</tr>
<tr>
<td>binary</td>
<td>ORC Format</td>
<td><code>read_orc</code></td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td><code>read_stata</code></td>
<td><code>to_stata</code></td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td><code>read_sas</code></td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>SPSS</td>
<td><code>read_spss</code></td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td><code>read_pickle</code></td>
<td><code>to_pickle</code></td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td><code>read_sql</code></td>
<td><code>to_sql</code></td>
</tr>
<tr>
<td>SQL</td>
<td>Google BigQuery</td>
<td><code>read_gbq</code></td>
<td><code>to_gbq</code></td>
</tr>
</tbody>
</table>

Here is an informal performance comparison for some of these IO methods.
Note: For examples that use the StringIO class, make sure you import it with `from io import StringIO` for Python 3.

## 2.4.1 CSV & text files

The workhorse function for reading text files (a.k.a. flat files) is `read_csv()`. See the cookbook for some advanced strategies.

### Parsing options

`read_csv()` accepts the following common arguments:

#### Basic

- **filepath_or_buffer** [various] Either a path to a file (a `str`, `pathlib.Path`, or `py:py._path.local.LocalPath`), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or `StringIO`).

- **sep** [str, defaults to ',' for `read_csv()`, \t for `read_table()`] Delimiter to use. If `sep` is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `\s+` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

- **delimiter** [str, default `None`] Alternative argument name for `sep`.

- **delim_whitespace** [boolean, default `False`] Specifies whether or not whitespace (e.g. `' '` or `\t`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

#### Column and index locations and names

- **header** [int or list of ints, default 'infer'] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names.

  The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. `[0, 1, 3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

- **names** [array-like, default `None`] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list are not allowed.

- **index_col** [int, str, sequence of int / str, or False, optional, default `None`] Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used. Note: `index_col=False` can be used to force pandas to _not_ use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.
The default value of None instructs pandas to guess. If the number of fields in the column header row is equal to the number of fields in the body of the data file, then a default index is used. If it is larger, then the first columns are used as index so that the remaining number of fields in the body are equal to the number of fields in the header.

The first row after the header is used to determine the number of columns, which will go into the index. If the subsequent rows contain less columns than the first row, they are filled with NaN.

This can be avoided through usecols. This ensures that the columns are taken as is and the trailing data are ignored.

**usecols** [list-like or callable, default None] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If names are given, the document header row(s) are not taken into account. For example, a valid list-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

Element order is ignored, so usecols=[0, 1] is the same as [1, 0]. To instantiate a DataFrame from data with element order preserved use pd.read_csv(data, usecols=[`foo`, 'bar'])[[`foo`, 'bar']] for columns in [`foo`, 'bar'] order or pd.read_csv(data, usecols=[`foo`, 'bar'])[[`bar`, 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```python
In [1]: import pandas as pd
In [2]: from io import StringIO
In [3]: data = "col1,col2,col3\n\n   a,b,1\n   a,b,2\n   c,d,3"
In [4]: pd.read_csv(StringIO(data))
Out[4]:
   col1  col2  col3
0    a    b    1
1    a    b    2
2    c    d    3
In [5]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["COL1", "COL3"])
Out[5]:
   col1  col3
0    a    1
1    a    2
2    c    3
```

Using this parameter results in much faster parsing time and lower memory usage when using the c engine. The Python engine loads the data first before deciding which columns to drop.

**squeeze** [boolean, default False] If the parsed data only contains one column then return a Series.

Deprecated since version 1.4.0: Append .squeeze("columns") to the call to {func_name} to squeeze the data.

**prefix** [str, default None] Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, …

Deprecated since version 1.4.0: Use a list comprehension on the DataFrame’s columns after calling read_csv.
In [6]: data = "col1,col2,col3\na,b,1"

In [7]: df = pd.read_csv(StringIO(data))

In [8]: df.columns = [f"pre_{col}" for col in df.columns]

In [9]: df
Out[9]:
   pre_col1  pre_col2  pre_col3
0       a       b        1

mangle_dupe_cols [boolean, default True] Duplicate columns will be specified as ‘X’, ‘X.1’…’X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

General parsing configuration

dtype [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (unsupported with engine='python'). Use str or object together with suitable na_values settings to preserve and not interpret dtype.

engine [{'c', 'python', 'pyarrow'}] Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

New in version 1.4.0: The “pyarrow” engine was added as an experimental engine, and some features are unsupported, or may not work correctly, with this engine.

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, default None] Values to consider as True.

false_values [list, default None] Values to consider as False.

skipinitialspace [boolean, default False] Skip spaces after delimiter.

skiprows [list-like or integer, default None] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

In [10]: data = "col1,col2,col3\na,b,1\nb,a,2\nc,d,3"

In [11]: pd.read_csv(StringIO(data))
Out[11]:
   col1  col2  col3
0   a    b    1
1   b    a    2
2   c    d    3

In [12]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[12]:
   col1  col2  col3
0   a    b    2

skipfooter [int, default 0] Number of lines at bottom of file to skip (unsupported with engine='c').
**nrows** [int, default None] Number of rows of file to read. Useful for reading pieces of large files.

**low_memory** [boolean, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser)

**memory_map** [boolean, default False] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

**NA and missing data handling**

**na_values** [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See na values const below for a list of the values interpreted as NaN by default.

**keep_default_na** [boolean, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether na_values is passed in, the behavior is as follows:

- If keep_default_na is True, and na_values are specified, na_values is appended to the default NaN values used for parsing.
- If keep_default_na is True, and na_values are not specified, only the default NaN values are used for parsing.
- If keep_default_na is False, and na_values are specified, only the NaN values specified na_values are used for parsing.
- If keep_default_na is False, and na_values are not specified, no strings will be parsed as NaN.

Note that if na_filter is passed in as False, the keep_default_na and na_values parameters will be ignored.

**na_filter** [boolean, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

**verbose** [boolean, default False] Indicate number of NA values placed in non-numeric columns.

**skip_blank_lines** [boolean, default True] If True, skip over blank lines rather than interpreting as NaN values.

**Datetime handling**

**parse_dates** [boolean or list of ints or names or list of lists or dict, default False.]

- If True -> try parsing the index.
- If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- If {'foo': [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’. A fast-path exists for iso8601-formatted dates.

**infer_datetime_format** [boolean, default False] If True and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

**keep_date_col** [boolean, default False] If True and parse_dates specifies combining multiple columns then keep the original columns.
**pandas**: powerful Python data analysis toolkit, Release 1.4.2

**date_parser** [function, default None] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call `date_parser` in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

**dayfirst** [boolean, default False] DD/MM format dates, international and European format.

**cache_dates** [boolean, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets. New in version 0.25.0.

**Iteration**

**iterator** [boolean, default False] Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

**chunksize** [int, default None] Return `TextFileReader` object for iteration. See iterating and chunking below.

**Quoting, compression, and file format**

**compression** [{'infer', 'gzip', 'bz2', 'zip', 'xz', 'zstd', None, dict}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip, xz, or zstandard if `filepath_or_buffer` is path-like ending in `.gz`, `.bz2`, `.zip`, `.xz`, `.zst`, respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key 'method' set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

Changed in version 1.1.0: dict option extended to support gzip and bz2.

Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to `gzip.open`.

**thousands** [str, default None] Thousands separator.

**decimal** [str, default ','] Character to recognize as decimal point. E.g. use ',' for European data.

**float_precision** [string, default None] Specifies which converter the C engine should use for floating-point values. The options are None for the ordinary converter, high for the high-precision converter, and round_trip for the round-trip converter.

**lineterminator** [str (length 1), default None] Character to break file into lines. Only valid with C parser.

**quoting** [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per csv.QUOTE_* constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

**doublequote** [boolean, default True] When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements inside a field as a single `quotechar` element.

**escapechar** [str (length 1), default None] One-character string used to escape delimiter when `quoting` is `QUOTE_NONE`.

**comment** [str, default None] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`.
For example, if `comment='#'`, parsing ‘#empty
a,b,c
1,2,3’ with `header=0` will result in ‘a,b,c’ being treated as the header.

**encoding** [str, default None] Encoding to use for UTF when reading/writing (e.g. 'utf-8'). List of Python standard encodings.

**dialect** [str or `csv.Dialect` instance, default None] If provided, this parameter will override values (default or not) for the following parameters: delimiter, doublequote, escapechar, skipinitialspace, quotechar, and quoting. If it is necessary to override values, a ParserWarning will be issued. See `csv.Dialect` documentation for more details.

### Error handling

**error_bad_lines** [boolean, optional, default None] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If False, then these “bad lines” will dropped from the `DataFrame` that is returned. See bad lines below.

Deprecated since version 1.3.0: The `on_bad_lines` parameter should be used instead to specify behavior upon encountering a bad line instead.

**warn_bad_lines** [boolean, optional, default None] If `error_bad_lines` is False, and `warn_bad_lines` is True, a warning for each “bad line” will be output.

Deprecated since version 1.3.0: The `on_bad_lines` parameter should be used instead to specify behavior upon encountering a bad line instead.

**on_bad_lines** [('error', 'warn', 'skip'), default 'error'] Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are:

- ‘error’, raise an `ParserError` when a bad line is encountered.
- ‘warn’, print a warning when a bad line is encountered and skip that line.
- ‘skip’, skip bad lines without raising or warning when they are encountered.

New in version 1.3.0.

### Specifying column data types

You can indicate the data type for the whole `DataFrame` or individual columns:

```python
In [13]: import numpy as np
In [14]: data = "a,b,c,d\n1,2,3,4\n5,6,7,8\n9,10,11"
In [15]: print(data)
a, b, c, d
1, 2, 3, 4
5, 6, 7, 8
9, 10, 11
In [16]: df = pd.read_csv(StringIO(data), dtype=object)
In [17]: df
Out[17]:
   a  b  c  d
0  1  2  3  4
```

(continues on next page)
Fortunately, pandas offers more than one way to ensure that your column(s) contain only one dtype. If you're unfamiliar with these concepts, you can see here to learn more about dtypes, and here to learn more about object conversion in pandas.

For instance, you can use the converters argument of read_csv():

```
In [21]: data = "col_1\n1\n2\n'A'\n4.22"

In [22]: df = pd.read_csv(StringIO(data), converters={"col_1": str})

In [23]: df
Out[23]:
   col_1
0    1
1    2
2    A
3  4.22

In [24]: df["col_1"].apply(type).value_counts()
Out[24]:
<class 'str'>    4
Name: col_1, dtype: int64
```

Or you can use the to_numeric() function to coerce the dtypes after reading in the data.

```
In [25]: df2 = pd.read_csv(StringIO(data))

In [26]: df2["col_1"] = pd.to_numeric(df2["col_1"], errors="coerce")

In [27]: df2
Out[27]:
   col_1
0    1.00
1    2.00
2    NaN
3    4.22
```
which will convert all valid parsing to floats, leaving the invalid parsing as NaN.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to NaN out the data anomalies, then to_numeric() is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the converters argument of read_csv() would certainly be worth trying.

Note: In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [29]: col_1 = list(range(500000)) + ['a', 'b'] + list(range(500000))
In [30]: df = pd.DataFrame({'col_1': col_1})
In [31]: df.to_csv('foo.csv')
In [32]: mixed_df = pd.read_csv('foo.csv')
In [33]: mixed_df['col_1'].apply(type).value_counts()
Out[33]:
<class 'int'>  737858
<class 'str'>  262144
Name: col_1, dtype: int64
In [34]: mixed_df['col_1'].dtype
Out[34]: dtype('O')
```

will result with mixed_df containing an int dtype for certain chunks of the column, and str for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a dtype of object, which is used for columns with mixed dtypes.

### Specifying categorical dtype

Categorical columns can be parsed directly by specifying dtype='category' or dtype=CategoricalDtype(categories, ordered).

```
In [35]: data = "col1, col2, col3\na, b, 1\na, b, 2\nc, d, 3"

In [36]: pd.read_csv(StringIO(data))
Out[36]:
   col1 col2  col3
0     a   b     1
1     a   b     2
```

pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

2

c

d

3

In [37]: pd.read_csv(StringIO(data)).dtypes
Out[37]:
col1
object
col2
object
col3
int64
dtype: object
In [38]: pd.read_csv(StringIO(data), dtype="category").dtypes
Out[38]:
col1
category
col2
category
col3
category
dtype: object
Individual columns can be parsed as a Categorical using a dict specification:
In [39]: pd.read_csv(StringIO(data), dtype={"col1": "category"}).dtypes
Out[39]:
col1
category
col2
object
col3
int64
dtype: object
Specifying dtype='category' will result in an unordered Categorical whose categories are the unique values
observed in the data. For more control on the categories and order, create a CategoricalDtype ahead of time, and
pass that for that column’s dtype.
In [40]: from pandas.api.types import CategoricalDtype
In [41]: dtype = CategoricalDtype(["d", "c", "b", "a"], ordered=True)
In [42]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).dtypes
Out[42]:
col1
category
col2
object
col3
int64
dtype: object
When using dtype=CategoricalDtype, “unexpected” values outside of dtype.categories are treated as missing
values.
In [43]: dtype = CategoricalDtype(["a", "b", "d"])

# No 'c'

In [44]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).col1
Out[44]:
0
a
1
a
2
NaN
Name: col1, dtype: category
Categories (3, object): ['a', 'b', 'd']

282

Chapter 2. User Guide


This matches the behavior of `Categorical.set_categories()`.

**Note:** With `dtype='category'`, the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the `to_numeric()` function, or as appropriate, another converter such as `to_datetime()`.

When `dtype` is a `CategoricalDtype` with homogeneous categories (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [45]: df = pd.read_csv(StringIO(data), dtype="category")
In [46]: df.dtypes
Out[46]:
col1    category
col2    category
col3    category
dtype: object

In [47]: df["col3"]
Out[47]:
0   1
1   2
2   3
Name: col3, dtypes: category
Categories (3, object): ['1', '2', '3']

In [48]: df["col3"].cat.categories = pd.to_numeric(df["col3"].cat.categories)
In [49]: df["col3"]
Out[49]:
0   1
1   2
2   3
Name: col3, dtypes: category
Categories (3, int64): [1, 2, 3]
```

### Naming and using columns

#### Handling column names

A file may or may not have a header row. `pandas` assumes the first row should be used as the column names:

```
In [50]: data = "a,b,c\n\n1,2,3\n4,5,6\n7,8,9"

In [51]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [52]: pd.read_csv(StringIO(data))
```

(continues on next page)
By specifying the names argument in conjunction with header you can indicate other names to use and whether or not to throw away the header row (if any):

```python
In [53]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9
```

```python
In [54]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[54]:
    foo  bar  baz
0   1    2    3
1   4    5    6
2   7    8    9
```

```python
In [55]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[55]:
    foo  bar  baz
0   a    b    c
1   1    2    3
2   4    5    6
3   7    8    9
```

If the header is in a row other than the first, pass the row number to header. This will skip the preceding rows:

```python
In [56]: data = "skip this skip it

a,b,c
1,2,3
4,5,6
7,8,9"
```

```python
In [57]: pd.read_csv(StringIO(data), header=1)
Out[57]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

**Note:** Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to header=None.
Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```python
In [58]: data = "a,b,a\n0,1,2\n3,4,5"

In [59]: pd.read_csv(StringIO(data))
Out[59]:
a  b  a.1
0  0  1  2
1  3  4  5
```

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns ‘X’, …, ‘X’ to become ‘X’, ‘X.1’, …, ‘X.N’. If `mangle_dupe_cols=False`, duplicate data can arise:

```python
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
a  b  a
0  2  1  2
1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```python
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...  
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

Filtering columns (usecols)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

```python
In [60]: data = "a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz"

In [61]: pd.read_csv(StringIO(data))
Out[61]:
a  b  c  d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [62]: pd.read_csv(StringIO(data), usecols=["b", "d"])
Out[62]:
b  d
0  2  foo
1  5  bar
2  8  baz
```

(continues on next page)
In [63]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[63]:
   a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [64]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
Out[64]:
   a  c
0  1  3
1  4  6
2  7  9

The usecols argument can also be used to specify which columns not to use in the final result:

In [65]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[65]:
   b  d
0  2  foo
1  5  bar
2  8  baz

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.

Comments and empty lines

Ignoring line comments and empty lines

If the comment parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

In [66]: data = "na,b,c

# commented line
1,2,3
4,5,6"

In [67]: print(data)
a,b,c
# commented line
1,2,3
4,5,6

In [68]: pd.read_csv(StringIO(data), comment='#')
Out[68]:
   a  b  c
0  1  2  3
1  4  5  6

If skip_blank_lines=False, then read_csv will not ignore blank lines:
In [69]: data = "a,b,c\n1,2,3\n4,5,6"

In [70]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[70]:
    a   b   c
0  NaN  NaN  NaN
1  1.0  2.0  3.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  4.0  5.0  6.0

Warning: The presence of ignored lines might create ambiguities involving line numbers; the parameter header uses row numbers (ignoring commented/empty lines), while skiprows uses line numbers (including commented/empty lines):

In [71]: data = "#comment
a,b,c
A,B,C
1,2,3"

In [72]: pd.read_csv(StringIO(data), comment="#", header=1)
Out[72]:
   A  B  C
0  1  2  3

In [73]: data = "A,B,C
#comment
a,b,c
1,2,3"

In [74]: pd.read_csv(StringIO(data), comment="#", skiprows=2)
Out[74]:
    a   b   c
0  1.0  2.0  3.0

If both header and skiprows are specified, header will be relative to the end of skiprows. For example:

In [75]: data = (  
...:     "# empty\n"  
...:     "# second empty line\n"  
...:     "# third emptyline\n"  
...:     "X,Y,Z\n"  
...:     "1,2,3\n"  
...:     "A,B,C\n"  
...:     "1.2.,4.\n"  
...:     "5.,NaN,10.0\n"
...: )

In [76]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1.2.,4.

(continues on next page)
In [77]: pd.read_csv(StringIO(data), comment="#", skiprows=4, header=1)

Out[77]:
      A   B   C
0  1.0  2.0  4.0
1  5.0   NaN 10.0

Comments

Sometimes comments or meta data may be included in a file:

In [78]: print(open("tmp.csv").read())

ID,level,category
Patient1,123000,x  # really unpleasant
Patient2,23000,y  # wouldn't take his medicine
Patient3,1234018,z  # awesome

By default, the parser includes the comments in the output:

In [79]: df = pd.read_csv("tmp.csv")

In [80]: df

Out[80]:
     ID   level   category
0 Patient1  123000         x  # really unpleasant
1 Patient2  23000        y  # wouldn't take his medicine
2 Patient3 1234018        z  # awesome

We can suppress the comments using the comment keyword:

In [81]: df = pd.read_csv("tmp.csv", comment="#")

In [82]: df

Out[82]:
     ID   level  category
0 Patient1  123000         x
1 Patient2  23000         y
2 Patient3 1234018        z

Dealing with Unicode data

The encoding argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

In [83]: from io import BytesIO

In [84]: data = b"word,length\n" b"Tr\xc3\xa4umen,7\n" b"Gr\xc3\xbc\xc3\xe9fe,5"

(continues on next page)
In [85]: data = data.decode("utf8").encode("latin-1")

In [86]: df = pd.read_csv(BytesIO(data), encoding="latin-1")

In [87]: df
Out[87]:
word length
0  Träumen  7
1   Grüße   5

In [88]: df["word"]
Out[88]: 'Grüße'

Some formats which encode all characters as multiple bytes, like UTF-16, won’t parse correctly at all without specifying the encoding. Full list of Python standard encodings.

**Index columns and trailing delimiters**

If a file has one more column of data than the number of column names, the first column will be used as the DataFrame’s row names:

In [89]: data = "a,b,c

In [90]: pd.read_csv(StringIO(data))
Out[90]:
   a  b  c
4  apple  bat  5.7
8  orange cow  10.0

In [91]: data = "index,a,b,c

In [92]: pd.read_csv(StringIO(data), index_col=0)
Out[92]:
   a  b  c
index
4  apple  bat  5.7
8  orange cow  10.0

Ordinarily, you can achieve this behavior using the index_col option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass index_col=False:

In [93]: data = "a,b,c

In [94]: print(data)
a,b,c
4,apple, bat, 
8,orange, cow,

In [95]: pd.read_csv(StringIO(data))
Out[95]:

(continues on next page)
If a subset of data is being parsed using the `usecols` option, the `index_col` specification is based on that subset, not the original data.

```
In [97]: data = "a,b,c
4,apple,bat,
8,orange,cow,"
```

```
In [98]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,
```

```
In [99]: pd.read_csv(StringIO(data), usecols=["b", "c"])
```

```
Out[99]:
   b  c
0  4  bat NaN
1  8  cow NaN
```

```
In [100]: pd.read_csv(StringIO(data), usecols=["b", "c"], index_col=0)
```

```
Out[100]:
   b  c
0  4  bat NaN
1  8  cow NaN
```

## Date Handling

### Specifying date columns

To better facilitate working with datetime data, `read_csv()` uses the keyword arguments `parse_dates` and `date_parser` to allow users to specify a variety of columns and date/time formats to turn the input text data into `datetime` objects.

The simplest case is to just pass in `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [101]: df = pd.read_csv("foo.csv", index_col=0, parse_dates=True)
```

```
In [102]: df
```

```
Out[102]:
     A  B  C
date
2009-01-01 a 1 2
2009-01-02 b 3 4
```
It is often the case that we may want to store date and time data separately, or store various date fields separately. The `parse_dates` keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to `parse_dates`, the resulting date columns will be prepended to the output (so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

```
In [104]: print(open("tmp.csv").read())
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900
```

```
In [105]: df = pd.read_csv("tmp.csv", header=None, parse_dates=[[1, 2], [1, 3]])
```

```
In [106]: df
Out[106]:
   1_2  1_3  0  1  2  3  4
0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [107]: df = pd.read_csv("tmp.csv", header=None, parse_dates=[[1, 2], [1, 3]], keep_date_col=True)
```

```
In [108]: df
Out[108]:
   1_2  1_3  0  1  2  3  4
0 1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 19990127 19:00:00 18:56:00 0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 19990127 20:00:00 19:56:00 0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00 KORD 19990127 21:00:00 20:56:00 -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00 KORD 19990127 21:00:00 21:18:00 -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00 KORD 19990127 22:00:00 21:56:00 -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00 KORD 19990127 23:00:00 22:56:00 -0.59
```

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, `parse_dates=[[1, 2]]` indicates that the second and third columns should each be parsed as separate date columns.
while parse_dates=[[1, 2]] means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [109]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}
In [110]: df = pd.read_csv("tmp.csv", header=None, parse_dates=date_spec)
In [111]: df
Out[111]:
   nominal  actual
0  1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 0.81
1  1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 0.01
2  1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
3  1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
4  1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
5  1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The index_col specification is based off of this new set of columns rather than the original data columns:

```
In [112]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}
In [113]: df = pd.read_csv(...:
   .....:   "tmp.csv", header=None, parse_dates=date_spec, index_col=0
   .....: ) # index is the nominal column
   .....:
In [114]: df
Out[114]:
    actual  0  4
   nominal
1999-01-27 19:00:00 1999-01-27 18:56:00 KORD 0.81
1999-01-27 20:00:00 1999-01-27 19:56:00 KORD 0.01
1999-01-27 21:00:00 1999-01-27 20:56:00 KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00 KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00 KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00 KORD -0.59
```

Note: If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `to_datetime()` after `pd.read_csv`.

Note: `read_csv` has a fast_path for parsing datetime strings in iso8601 format, e.g "2000-01-01T00:01:02+00:00" and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.
Date parsing functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```python
In [115]: df = pd.read_csv(
    .....:     "tmp.csv", header=None, parse_dates=date_spec, date_parser=pd.to_datetime
    .....:     )
```

```python
In [116]: df
```

<table>
<thead>
<tr>
<th></th>
<th>nominal</th>
<th>actual</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1999-01-27 19:00:00</td>
<td>1999-01-27 18:56:00</td>
<td>KORD 0.81</td>
</tr>
<tr>
<td>1</td>
<td>1999-01-27 20:00:00</td>
<td>1999-01-27 19:56:00</td>
<td>KORD 0.01</td>
</tr>
<tr>
<td>2</td>
<td>1999-01-27 21:00:00</td>
<td>1999-01-27 20:56:00</td>
<td>KORD -0.59</td>
</tr>
<tr>
<td>3</td>
<td>1999-01-27 21:00:00</td>
<td>1999-01-27 21:18:00</td>
<td>KORD -0.99</td>
</tr>
<tr>
<td>4</td>
<td>1999-01-27 22:00:00</td>
<td>1999-01-27 21:56:00</td>
<td>KORD -0.59</td>
</tr>
<tr>
<td>5</td>
<td>1999-01-27 23:00:00</td>
<td>1999-01-27 22:56:00</td>
<td>KORD -0.59</td>
</tr>
</tbody>
</table>

Pandas will try to call the `date_parser` function in three different ways. If an exception is raised, the next one is tried:

1. `date_parser` is first called with one or more arrays as arguments, as defined using `parse_dates` (e.g., `date_parser(['2013', '2013'], ['1', '2'])`).
2. If #1 fails, `date_parser` is called with all the columns concatenated row-wise into a single array (e.g., `date_parser(['2013 1', '2013 2'])`).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using `infer_datetime_format=True` (see section below).
2. If you know the format, use `pd.to_datetime()`: `date_parser=lambda x: pd.to_datetime(x, format=...)`.
3. If you have a really non-standard format, use a custom `date_parser` function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

Parsing a CSV with mixed timezones

Pandas cannot natively represent a column or index with mixed timezones. If your CSV file contains columns with a mixture of timezones, the default result will be an object-dtype column with strings, even with `parse_dates`.

```python
In [117]: content = ""
    .....:     """""""""""""
    .....:     a
    .....:     2000-01-01T00:00:00+05:00
    .....:     2000-01-01T00:00:00+06:00"
    .....:     """

In [118]: df = pd.read_csv(StringIO(content), parse_dates=['a'])
```

```python
In [119]: df['a']
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2000-01-01 00:00:00+05:00</td>
</tr>
<tr>
<td>1</td>
<td>2000-01-01 00:00:00+06:00</td>
</tr>
</tbody>
</table>

Name: a, dtype: object
To parse the mixed-timezone values as a datetime column, pass a partially-applied `to_datetime()` with `utc=True` as the `date_parser`.

```
In [120]: df = pd.read_csv(
    ....:     StringIO(content),
    ....:     parse_dates=['a'],
    ....:     date_parser=lambda col: pd.to_datetime(col, utc=True),
    ....: )
```

```
In [121]: df['a']
Out[121]:
0  1999-12-31 19:00:00+00:00
1  1999-12-31 18:00:00+00:00
Name: a, dtype: datetime64[ns, UTC]
```

### Inferring datetime format

If you have `parse_dates` enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting `infer_datetime_format=True`. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, `infer_datetime_format` should not have any negative consequences if enabled.

Here are some examples of datetime strings that can be guessed (All representing December 30th, 2011 at 00:00:00):

- “20111230”
- “2011/12/30”
- “20111230 00:00:00”
- “12/30/2011 00:00:00”
- “30/Dec/2011 00:00:00”
- “30/December/2011 00:00:00”

Note that `infer_datetime_format` is sensitive to `dayfirst`. With `dayfirst=True`, it will guess “01/12/2011” to be December 1st. With `dayfirst=False` (default) it will guess “01/12/2011” to be January 12th.

```
# Try to infer the format for the index column
In [122]: df = pd.read_csv(
    ....:     "foo.csv",
    ....:     index_col=0,
    ....:     parse_dates=True,
    ....:     infer_datetime_format=True,
    ....: )
```

```
In [123]: df
Out[123]:
          A  B  C
date
2009-01-01  a  1  2
```
International date formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a dayfirst keyword is provided:

```python
In [124]: print(open("tmp.csv").read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [125]: pd.read_csv("tmp.csv", parse_dates=[0])
Out[125]:
     date  value  cat
0 2000-01-06   5    a
1 2000-02-06  10    b
2 2000-03-06  15    c

In [126]: pd.read_csv("tmp.csv", dayfirst=True, parse_dates=[0])
Out[126]:
     date  value  cat
0     6 2000    5    a
1     6 2000    10   b
2     6 2000    15   c
```

Writing CSVs to binary file objects

New in version 1.2.0.

df.to_csv(..., mode="wb") allows writing a CSV to a file object opened binary mode. In most cases, it is not necessary to specify mode as Pandas will auto-detect whether the file object is opened in text or binary mode.

```python
In [127]: import io

In [128]: data = pd.DataFrame([0, 1, 2])

In [129]: buffer = io.BytesIO()

In [130]: data.to_csv(buffer, encoding="utf-8", compression="gzip")
```
Specifying method for floating-point conversion

The parameter `float_precision` can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```python
In [131]: val = "0.306610199380709547156698135950136927504425048828125"

In [132]: data = "a,b,c\n1,2,{0}".format(val)

In [133]: abs(pd.read_csv(StringIO(data), engine="c", float_precision=None)["c"][0] - float(val))

Out[133]: 5.551115123125783e-17

In [134]: abs(pd.read_csv(StringIO(data), engine="c", float_precision="high", fmt=windows	)["c"][0] - float(val))

Out[134]: 5.551115123125783e-17

In [135]: abs(pd.read_csv(StringIO(data), engine="c", float_precision="round_trip")["c\t"])[0] - float(val))

Out[135]: 0.0
```

Thousand separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```python
In [136]: print(open("tmp.csv").read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [137]: df = pd.read_csv("tmp.csv", sep="|")
```

(continues on next page)
The thousands keyword allows integers to be parsed correctly:

```python
In [140]: print(open("tmp.csv").read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z
```

```python
In [141]: df = pd.read_csv("tmp.csv", sep="|", thousands="",)
```

```python
In [142]: df
Out[142]:
   ID    level category
0  0   Patient1   123,000    x
1  1  Patient2    23,000    y
2  2  Patient3  1,234,018    z
```

### NA values

To control which values are parsed as missing values (which are signified by NaN), specify a string in `na_values`. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0, 5] are recognized as NaN).

To completely override the default values that are recognized as missing, specify `keep_default_na=False`.

The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'n/a', 'NA', '<NA>', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', ''].

Let us consider some examples:

```python
pd.read_csv("path_to_file.csv", na_values=[5])
```

In the example above 5 and 5.0 will be recognized as NaN, in addition to the defaults. A string will first be interpreted as a numerical 5, then as a NaN.

```python
pd.read_csv("path_to_file.csv", keep_default_na=False, na_values=[''])
```

Above, only an empty field will be recognized as NaN.

2.4. IO tools (text, CSV, HDF5, …)
Above, both NA and 0 as strings are NaN.

The default values, in addition to the string "Nope" are recognized as NaN.

**Infinity**

Inf like values will be parsed as np.inf (positive infinity), and -inf as -np.inf (negative infinity). These will ignore the case of the value, meaning Inf, will also be parsed as np.inf.

**Returning Series**

Using the `squeeze` keyword, the parser will return output with a single column as a Series:

Deprecated since version 1.4.0: Users should append `.squeeze("columns")` to the DataFrame returned by `read_csv` instead.

```python
In [144]: print(open("tmp.csv").read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [145]: output = pd.read_csv("tmp.csv", squeeze=True)

In [146]: output
Out[146]:
      Patient1
level
Patient1    123000
Patient2    23000
Patient3    1234018
Name: level, dtype: int64

In [147]: type(output)
Out[147]: pandas.core.series.Series
```

**Boolean values**

The common values True, False, TRUE, and FALSE are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the `true_values` and `false_values` options as follows:

```python
In [148]: data = "a,b,c\n1,Yes,2\n3,No,4"

In [149]: print(data)
a,b,c
1,Yes,2
3,No,4

In [150]: pd.read_csv(StringIO(data))
```
Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many fields will raise an error by default:

```
In [152]: data = "a,b,c

1,2,3
4,5,6,7
8,9,10"

In [153]: pd.read_csv(StringIO(data))
```

```
ParserError Traceback (most recent call last)
Input In [153], in <cell line: 1>()
----> 1 pd.read_csv(StringIO(data))

File /pandas/pandas/util/_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>.wrapper(*args, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310         )
--> 311     return func(*args, **kwargs)

File /pandas/pandas/io/parsers/readers.py:680, in read_csv(filepath_or_buffer, sep,...
    665 kwds_defaults = _refine_defaults_read(
    666     dialect,
    667     (,...
    676     defaults={"delimiter": ","},
    677     )
    678 kwds.update(kwds_defaults)
--> 680 return _read(filepath_or_buffer, kwds)
```

(continues on next page)
You can elect to skip bad lines:

```python
In [29]: pd.read_csv(StringIO(data), on_bad_lines="warn")
Skipping line 3: expected 3 fields, saw 4
```

```plaintext
Out[29]:
a  b  c
0 1 2 3
1 8 9 10
```

Or pass a callable function to handle the bad line if engine="python". The bad line will be a list of strings that was split by the sep:

```python
In [29]: external_list = []
In [30]: def bad_lines_func(line):
   ...:     external_list.append(line)
   ...:     return line[-3:]
```
In [31]: pd.read_csv(StringIO(data), on_bad_lines=bad_lines_func, engine="python")
Out[31]:
   a  b  c
0  1  2  3
1  5  6  7
2  8  9 10

In [32]: external_list
Out[32]: [4, 5, 6, 7]

.. versionadded:: 1.4.0

You can also use the usecols parameter to eliminate extraneous column data that appear in some lines but not others:

In [33]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

Out[33]:
   a  b  c
0  1  2  3
1  4  5  6
2  8  9 10

In case you want to keep all data including the lines with too many fields, you can specify a sufficient number of names. This ensures that lines with not enough fields are filled with NaN.

In [34]: pd.read_csv(StringIO(data), names=['a', 'b', 'c', 'd'])

Out[34]:
   a  b  c  d
0  1  2  3  NaN
1  4  5  6  NaN
2  8  9 10  NaN

Dialect

The dialect keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a csv.Dialect instance.

Suppose you had data with unenclosed quotes:

In [154]: print(data)
label1,label2,label3
index1,"a,c,e
index2,"b,d,f

By default, read_csv uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using dialect:

In [155]: import csv

(continues on next page)
In [156]: dia = csv.excel()

In [157]: dia.quoting = csv.QUOTE_NONE

In [158]: pd.read_csv(StringIO(data), dialect=dia)
   Out[158]:
       label1  label2  label3
   index1   "a  c  e
   index2   b  d  f

All of the dialect options can be specified separately by keyword arguments:

In [159]: data = "a,b,c~1,2,3~4,5,6"

In [160]: pd.read_csv(StringIO(data), lineterminator="~")
   Out[160]:
    a  b  c
   0  1  2  3
   1  4  5  6

Another common dialect option is skipinitialspace, to skip any whitespace after a delimiter:

In [161]: data = "a, b, c\n, 2, 3\n, 4, 5, 6"

In [162]: print(data)
   a, b, c
   1, 2, 3
   4, 5, 6

In [163]: pd.read_csv(StringIO(data), skipinitialspace=True)
   Out[163]:
    a  b  c
   0  1  2  3
   1  4  5  6

The parsers make every attempt to “do the right thing” and not be fragile. Type inference is a pretty big deal. If a
column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns
will come through as object dtype as with the rest of pandas objects.

**Quoting and Escape Characters**

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use
backslashes; to properly parse this data, you should pass the escapechar option:

In [164]: data = 'a,b\n"hello, \"Bob\"", nice to see you",5'

In [165]: print(data)
   a,b
   "hello, "Bob\", nice to see you",5

In [166]: pd.read_csv(StringIO(data), escapechar="\")
   Out[166]:
(continues on next page)
Files with fixed width columns

While `read_csv()` reads delimited data, the `read_fwf()` function works with data files that have known and fixed column widths. The function parameters to `read_fwf` are largely the same as `read_csv` with two extra parameters, and a different usage of the `delimiter` parameter:

- **colspecs**: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., `[from, to[`). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.

- **widths**: A list of field widths which can be used instead of ‘colspecs’ if the intervals are contiguous.

- **delimiter**: Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., ‘~’).

Consider a typical fixed-width data file:

```
In [167]: print(open("bar.csv").read())
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

In order to parse this file into a DataFrame, we simply need to supply the column specifications to the `read_fwf` function along with the file name:

```
# Column specifications are a list of half-intervals
In [168]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [169]: df = pd.read_fwf("bar.csv", colspecs=colspecs, header=None, index_col=0)

In [170]: df
Out[170]:
          0         1         2         3
0 id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
# Widths are a list of integers
In [171]: widths = [6, 14, 13, 10]

In [172]: df = pd.read_fwf("bar.csv", widths=widths, header=None)
```

(continues on next page)
In [173]: df
Out[173]:
   0  1   2    3
0 id8141 360.242940 149.910199 11950.7
1 id1594 444.953632 166.985655 11788.4
2 id1849 364.136849 183.628767 11806.2
3 id1230 413.836124 184.375703 11916.8
4 id1948 502.953953 173.237159 12468.3

The parser will take care of extra white spaces around the columns so it’s ok to have extra separation between the columns in the file.

By default, read_fwf will try to infer the file’s colspecs by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided delimiter (default delimiter is whitespace).

In [174]: df = pd.read_fwf("bar.csv", header=None, index_col=0)
In [175]: df
Out[175]:
     1   2    3
0  id8141 360.242940 149.910199 11950.7
1  id1594 444.953632 166.985655 11788.4
2  id1849 364.136849 183.628767 11806.2
3  id1230 413.836124 184.375703 11916.8
4  id1948 502.953953 173.237159 12468.3

read_fwf supports the dtype parameter for specifying the types of parsed columns to be different from the inferred type.

In [176]: pd.read_fwf("bar.csv", header=None, index_col=0).dtypes
Out[176]:
    1       2       3
dtype: object
In [177]: pd.read_fwf("bar.csv", header=None, dtype={2: "object"}).dtypes
Out[177]:
    0       1       2       3
dtype: object
Indexes

Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

```python
In [178]: print(open("foo.csv").read())
A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

In this special case, `read_csv` assumes that the first column is to be used as the index of the DataFrame:

```python
In [179]: pd.read_csv("foo.csv")
Out[179]:
    A  B  C
0 20090101 a 1 2
1 20090102 b 3 4
2 20090103 c 4 5
```

Note that the dates weren’t automatically parsed. In that case you would need to do as before:

```python
In [180]: df = pd.read_csv("foo.csv", parse_dates=True)
In [181]: df.index
Out[181]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[ns']
```

Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

```python
In [182]: print(open("data/mindex_ex.csv").read())
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

The `index_col` argument to `read_csv` can take a list of column numbers to turn multiple columns into a MultiIndex for the index of the returned object:
In [183]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [184]: df
Out[184]:
   zit  xit
year indiv
1977  A  1.20  0.60
     B  1.50  0.50
     C  1.70  0.80
1978  A  0.20  0.06
     B  0.70  0.20
     C  0.80  0.30
     D  0.90  0.50
     E  1.40  0.90
1979  C  0.20  0.15
     D  0.14  0.05
     E  0.50  0.15
     F  1.20  0.50
     G  3.40  1.90
     H  5.40  2.70
     I  6.40  1.20

In [185]: df.loc[1978]
Out[185]:
   zit  xit
indiv
  A  0.2  0.06
  B  0.7  0.20
  C  0.8  0.30
  D  0.9  0.50
  E  1.4  0.90

Reading columns with a MultiIndex

By specifying list of row locations for the header argument, you can read in a MultiIndex for the columns. Specifying non-consecutive rows will skip the intervening rows.

In [186]: from pandas._testing import makeCustomDataframe as mkdf

In [187]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [188]: df.to_csv("mi.csv")

In [189]: print(open("mi.csv").read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

R_l0_g2, R_l1_g2, R2C0, R2C1, R2C2
R_l0_g3, R_l1_g3, R3C0, R3C1, R3C2
R_l0_g4, R_l1_g4, R4C0, R4C1, R4C2

In [190]: pd.read_csv("mi.csv", header=[0, 1, 2, 3], index_col=[0, 1])
Out[190]:

<table>
<thead>
<tr>
<th></th>
<th>C0</th>
<th>C_l0_g0</th>
<th>C_l0_g1</th>
<th>C_l0_g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C_l1_g0</td>
<td>C_l1_g1</td>
<td>C_l1_g2</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>C_l2_g0</td>
<td>C_l2_g1</td>
<td>C_l2_g2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>C_l3_g0</td>
<td>C_l3_g1</td>
<td>C_l3_g2</td>
<td></td>
</tr>
<tr>
<td>R0</td>
<td>R_l0_g0</td>
<td>R0C0</td>
<td>R0C1</td>
<td>R0C2</td>
</tr>
<tr>
<td>R1</td>
<td>R_l1_g0</td>
<td>R1C0</td>
<td>R1C1</td>
<td>R1C2</td>
</tr>
<tr>
<td>R_l0_g1</td>
<td>R_l1_g1</td>
<td>R2C0</td>
<td>R2C1</td>
<td>R2C2</td>
</tr>
<tr>
<td>R_l0_g2</td>
<td>R_l1_g2</td>
<td>R3C0</td>
<td>R3C1</td>
<td>R3C2</td>
</tr>
<tr>
<td>R_l0_g3</td>
<td>R_l1_g3</td>
<td>R4C0</td>
<td>R4C1</td>
<td>R4C2</td>
</tr>
</tbody>
</table>

read_csv is also able to interpret a more common format of multi-columns indices.

In [191]: print(open("mi2.csv").read())

```
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12
```

In [192]: pd.read_csv("mi2.csv", header=[0, 1], index_col=0)
Out[192]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>one,1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>two,7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Note: If an index_col is not specified (e.g. you don’t have an index, or wrote it with df.to_csv(..., index=False), then any names on the columns index will be lost.

Automatically “sniffing” the delimiter

read_csv is capable of inferring delimited (not necessarily comma-separated) files, as pandas uses the csv.Sniffer class of the csv module. For this, you have to specify sep=None.

In [193]: print(open("tmp2.sv").read())

```
0:0:1:2:3
0:0:0.4691122999071863:0.282633443286633:1.5090585031735124:1.35632371017934
1:1.2121120250208506:0.17321464905330858:0.11920871129693428:0.10442359662799567
2:0.861848963347999:2.1045692188948086:0.4949292740687813:0.71803080737338
3:0.7215551622443669:0.7067711336300845:1.0395749851146963:0.27185988554282986
4:0.4249723297883753:0.567020349793672:0.27623201297771873:0.874006912691915
5:0.673689708083706:0.113648809688855:1.4784265524372235:0.5249876671147047
6:0.404705218682365:0.5770459859204836:1.7150020161146375:1.0392684835147225
7:0.390646858236446:1.1578922506419993:1.344311812731667:0.8448851414248841
8:1.075697837155533:0.1090499752802223:1.643563073622064:1.4693879595399115
```

(continues on next page)

2.4. IO tools (text, CSV, HDF5, …) 307
In [194]: pd.read_csv("tmp2.sv", sep=None, engine="python")

Out[194]:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>1</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>3</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>4</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td>5</td>
<td>-0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>6</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td>7</td>
<td>-0.370647</td>
<td>-1.157892</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
<tr>
<td>8</td>
<td>1.075770</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>-1.469388</td>
</tr>
<tr>
<td>9</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>-1.776904</td>
<td>-0.968914</td>
</tr>
</tbody>
</table>

Reading multiple files to create a single DataFrame

It’s best to use `concat()` to combine multiple files. See the cookbook for an example.

Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

In [195]: print(open("tmp.sv").read())

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>1</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>3</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>4</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td>5</td>
<td>-0.673689</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>6</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td>7</td>
<td>-0.370646</td>
<td>-1.157892</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
<tr>
<td>8</td>
<td>1.075770</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>-1.469388</td>
</tr>
<tr>
<td>9</td>
<td>0.357020</td>
<td>-0.674600</td>
<td>-1.776904</td>
<td>-0.968914</td>
</tr>
</tbody>
</table>

In [196]: table = pd.read_csv("tmp.sv", sep="\|")

In [197]: table

Out[197]:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>1</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>3</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>4</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
</tbody>
</table>

(continues on next page)
By specifying a chunksize to read_csv, the return value will be an iterable object of type TextFileReader:

```
In [198]: with pd.read_csv("tmp.sv", sep="|", chunksize=4) as reader:
    ...:     for chunk in reader:
    ...:         print(chunk)
```

```
    Unnamed: 0   0   1  2  3
     0  0 0.469112 -0.282863 -1.509059 -1.135632
     1  1 1.212112 -0.173215  0.119209 -1.044236
     2  2 -0.861849 -2.104569 -0.494929  1.071804
     3  3 0.721555 -0.706771 -1.039575  0.271860
    Unnamed: 0   0   1  2  3
     4  4 -0.424972  0.567020  0.276232 -1.087401
     5  5 -0.673690  0.113648 -1.478427  0.524988
     6  6 0.404705  0.577046 -1.715002 -1.039268
     7  7 -0.370647 -1.157892 -1.344312  0.844885
    Unnamed: 0   0   1  2  3
     8  8 1.075770 -0.109050  1.643563 -1.469388
     9  9 0.357021 -0.674600 -1.776904 -0.968914
```

Changed in version 1.2: read_csv/json/sas return a context-manager when iterating through a file.

Specifying iterator=True will also return the TextFileReader object:

```
In [199]: with pd.read_csv("tmp.sv", sep="|", iterator=True) as reader:
    ...:     reader.get_chunk(5)
```

### Specifying the parser engine

Pandas currently supports three engines, the C engine, the python engine, and an experimental pyarrow engine (requires the pyarrow package). In general, the pyarrow engine is fastest on larger workloads and is equivalent in speed to the C engine on most other workloads. The python engine tends to be slower than the pyarrow and C engines on most workloads. However, the pyarrow engine is much less robust than the C engine, which lacks a few features compared to the Python engine.

Where possible, pandas uses the C parser (specified as engine=’c’), but it may fall back to Python if C-unsupported options are specified.

Currently, options unsupported by the C and pyarrow engines include:

- `sep` other than a single character (e.g. regex separators)
- `skipfooter`
- `sep=None with delim_whitespace=False`
Specifying any of the above options will produce a `ParserWarning` unless the python engine is selected explicitly using `engine='python'`.

Options that are unsupported by the pyarrow engine which are not covered by the list above include:

- `float_precision`
- `chunksize`
- `comment`
- `nrows`
- `thousands`
- `memory_map`
- `dialect`
- `warn_bad_lines`
- `error_bad_lines`
- `on_bad_lines`
- `delim_whitespace`
- `quoting`
- `lineterminator`
- `converters`
- `decimal`
- `iterator`
- `dayfirst`
- `infer_datetime_format`
- `verbose`
- `skipinitialspace`
- `low_memory`

Specifying these options with `engine='pyarrow'` will raise a `ValueError`.

**Reading/writing remote files**

You can pass in a URL to read or write remote files to many of pandas' IO functions - the following example shows reading a CSV file:

```python
df = pd.read_csv("https://download.bls.gov/pub/time.series/cu/cu.item", sep="\t")
```

New in version 1.3.0.

A custom header can be sent alongside HTTP(s) requests by passing a dictionary of header key value mappings to the `storage_options` keyword argument as shown below:

```python
headers = {"User-Agent": "pandas"}
df = pd.read_csv(
    "https://download.bls.gov/pub/time.series/cu/cu.item",
    sep="\t",
    storage_options=headers,
)
```
All URLs which are not local files or HTTP(s) are handled by fsspec, if installed, and its various filesystem implementations (including Amazon S3, Google Cloud, SSH, FTP, webHDFS...). Some of these implementations will require additional packages to be installed, for example S3 URLs require the s3fs library:

```python
storage_options=headers
)
```

When dealing with remote storage systems, you might need extra configuration with environment variables or config files in special locations. For example, to access data in your S3 bucket, you will need to define credentials in one of the several ways listed in the S3Fs documentation. The same is true for several of the storage backends, and you should follow the links at fSIMPl1 for implementations built into fsspec and fSIMPl2 for those not included in the main fsspec distribution.

You can also pass parameters directly to the backend driver. For example, if you do not have S3 credentials, you can still access public data by specifying an anonymous connection, such as

```python
New in version 1.2.0.

def = pd.read_json("s3://pandas-test/adatafile.json")
```

fsspec also allows complex URLs, for accessing data in compressed archives, local caching of files, and more. To locally cache the above example, you would modify the call to

```python
fsspec also allows complex URLs, for accessing data in compressed archives, local caching of files, and more. To locally cache the above example, you would modify the call to

```python
pd.read_csv("simplecache::s3://ncei-wcsd-archive/data/processed/SH1305/18kHz/"
"SaKe2013-D20130523-T080854_to_SaKe2013-D20130523-T085643.csv",
storage_options={"s3": {"anon": True}},
)
```

where we specify that the “anon” parameter is meant for the “s3” part of the implementation, not to the caching implementation. Note that this caches to a temporary directory for the duration of the session only, but you can also specify a permanent store.

### Writing out data

#### Writing to CSV format

The `Series` and `DataFrame` objects have an instance method `to_csv` which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- **path_or_buf**: A string path to the file to write or a file object. If a file object it must be opened with `newline=''
- **sep**: Field delimiter for the output file (default ”,”)
- **na_rep**: A string representation of a missing value (default ””)
- **float_format**: Format string for floating point numbers
- **columns**: Columns to write (default None)
header: Whether to write out the column names (default True)

index: whether to write row (index) names (default True)

index_label: Column label(s) for index column(s) if desired. If None (default), and header and index are
True, then the index names are used. (A sequence should be given if the DataFrame uses MultiIndex).

mode : Python write mode, default ‘w’

encoding: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3

line_terminator: Character sequence denoting line end (default os.linesep)

quoting: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a
float_format then floats are converted to strings and csv.QUOTE_NONNUMERIC will treat them as non-
numeric

quotechar: Character used to quote fields (default ‘”’)

doublequote: Control quoting of quotechar in fields (default True)

escapechar: Character used to escape sep and quotechar when appropriate (default None)

chunksize: Number of rows to write at a time

date_format: Format string for datetime objects

Writing a formatted string

The DataFrame object has an instance method to_string which allows control over the string representation of the
object. All arguments are optional:

buf default None, for example a StringIO object

columns default None, which columns to write

col_space default None, minimum width of each column.

na_rep default NaN, representation of NA value

formatters default None, a dictionary (by column) of functions each of which takes a single argument and
returns a formatted string

float_format default None, a function which takes a single (float) argument and returns a formatted string; to
be applied to floats in the DataFrame.

sparsify default True, set to False for a DataFrame with a hierarchical index to print every MultiIndex key at
each row.

index_names default True, will print the names of the indices

index default True, will print the index (ie, row labels)

header default True, will print the column labels

justify default left, will print column headers left- or right-justified

The Series object also has a to_string method, but with only the buf, na_rep, float_format arguments. There
is also a length argument which, if set to True, will additionally output the length of the Series.
2.4.2 JSON

Read and write JSON format files and strings.

Writing JSON

A Series or DataFrame can be converted to a valid JSON string. Use `to_json` with optional parameters:

- **path_or_buf**: the pathname or buffer to write the output. This can be `None` in which case a JSON string is returned.
- **orient**:
  - *Series*:
    - default is `index`
    - allowed values are {split, records, index}
  - *DataFrame*:
    - default is `columns`
    - allowed values are {split, records, index, columns, values, table}

The format of the JSON string

<table>
<thead>
<tr>
<th>format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td>dict like <code>index -&gt; [index], columns -&gt; [columns], data -&gt; [values]</code></td>
</tr>
<tr>
<td>records</td>
<td>list like <code>[{column -&gt; value}, ... , {column -&gt; value}]</code></td>
</tr>
<tr>
<td>index</td>
<td>dict like <code>index -&gt; {column -&gt; value}</code></td>
</tr>
<tr>
<td>columns</td>
<td>dict like <code>{column -&gt; [index -&gt; value]}</code></td>
</tr>
<tr>
<td>values</td>
<td>just the values array</td>
</tr>
<tr>
<td>table</td>
<td>adhering to the JSON Table Schema</td>
</tr>
</tbody>
</table>

- **date_format**: string, type of date conversion, ‘epoch’ for timestamp, ‘iso’ for ISO8601.
- **double_precision**: The number of decimal places to use when encoding floating point values, default 10.
- **force_ascii**: force encoded string to be ASCII, default True.
- **date_unit**: The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’ or ‘ns’ for seconds, milliseconds, microseconds and nanoseconds respectively. Default ‘ms’.
- **default_handler**: The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.
- **lines**: If records orient, then will write each record per line as json.

Note *NaT*’s, *NaN*’s and *None* will be converted to null and datetime objects will be converted based on the `date_format` and `date_unit` parameters.

```python
In [200]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))
In [201]: json = dfj.to_json()
In [202]: json
Out[202]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":0.8957173022,"4":0.8052440254},"B":{"0":0.4137381054,"1":-0.472034511,"2":-0.9230545732, "3":0.8052440254,"4":0.8052440254}}'
```
Orient options

There are a number of different options for the format of the resulting JSON file / string. Consider the following DataFrame and Series:

```python
In [203]: dfjo = pd.DataFrame(
    ....:     dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
    ....:     columns=list("ABC"),
    ....:     index=list("xyz"),
    ....: )

In [204]: dfjo
Out[204]:
    A  B  C
  x  1  4  7
  y  2  5  8
  z  3  6  9

In [205]: sjo = pd.Series(dict(x=15, y=16, z=17), name="D")

In [206]: sjo
Out[206]:
     x    y    z
Name: D, dtype: int64
```

**Column oriented** (the default for DataFrame) serializes the data as nested JSON objects with column labels acting as the primary index:

```python
In [207]: dfjo.to_json(orient="columns")
Out[207]:
'{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'
```

# Not available for Series

**Index oriented** (the default for Series) similar to column oriented but the index labels are now primary:

```python
In [208]: dfjo.to_json(orient="index")
Out[208]:
'{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [209]: sjo.to_json(orient="index")
Out[209]:
'"x":15,"y":16,"z":17'
```

**Record oriented** serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing DataFrame data to plotting libraries, for example the JavaScript library `d3.js`:

```python
In [210]: dfjo.to_json(orient="records")
Out[210]:

In [211]: sjo.to_json(orient="records")
Out[211]:
'[15,16,17]'
```

**Value oriented** is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:
In [212]: dfjo.to_json(orient="values")
Out[212]: '[[1,4,7],[2,5,8],[3,6,9]]'

# Not available for Series

Split oriented serializes to a JSON object containing separate entries for values, index and columns. Name is also included for Series:

In [213]: dfjo.to_json(orient="split")
Out[213]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,9]]}'

In [214]: sjo.to_json(orient="split")
Out[214]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'

Table oriented serializes to the JSON Table Schema, allowing for the preservation of metadata including but not limited to dtypes and index names.

Note: Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the split option as it uses ordered containers.

Date handling

Writing in ISO date format:

In [215]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))
In [216]: dfd["date"] = pd.Timestamp("20130101")
In [217]: dfd = dfd.sort_index(axis=1, ascending=False)
In [218]: json = dfd.to_json(date_format="iso")
In [219]: json
Out[219]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":...","4":"2013-01-01T00:00:00.000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.4108345112,"4":0.1320031703}}'

Writing in ISO date format, with microseconds:

In [220]: json = dfd.to_json(date_format="iso", date_unit="us")
In [221]: json
Out[221]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z","2":...","4":0.8273169356},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.4108345112,"4":0.1320031703}}'

Epoch timestamps, in seconds:

In [222]: json = dfd.to_json(date_format="iso", date_unit="us")
In [223]: json
Out[223]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z","2":...","4":0.8273169356},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.4108345112,"4":0.1320031703}}'
In [222]: json = dfd.to_json(date_format="epoch", date_unit="s")

In [223]: json
Out[223]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":
->1356998400},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,
->"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.
->4108345112,"4":0.1320031703}}'

Writing to a file, with a date index and a date column:

In [224]: dfj2 = dfj.copy()
In [225]: dfj2['date'] = pd.Timestamp("20130101")
In [226]: dfj2['ints'] = list(range(5))
In [227]: dfj2['bools'] = True
In [228]: dfj2.index = pd.date_range("20130101", periods=5)
In [229]: dfj2.to_json("test.json")
In [230]: with open("test.json") as fh:
......:   print(fh.read())
......:}

Fallback behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. np.complex_) then the default_handler, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a toDict method and call it. A toDict method should return a dict which will then be JSON serialized.
  - invoke the default_handler if one was provided.
  - convert the object to a dict by traversing its contents. However this will often fail with an OverflowError or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a default_handler. For example:
can be dealt with by specifying a simple default_handler:

```python
In [231]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[231]: '{"0":{(1+0j)},"1":{(2+0j)},"2":{(1+2j)}}'
```

### Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a DataFrame if `typ` is not supplied or is `None`. To explicitly force Series parsing, pass `typ=series`

- **filepath_or_buffer**: a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, S3, and file. For file URLs, a host is expected. For instance, a local file could be file://localhost/path/to/table.json

- **typ**: type of object to recover (series or frame), default ‘frame’

- **orient**:

  - **Series**:
    - default is `index`
    - allowed values are `{split, records, index}`

  - **DataFrame**:
    - default is `columns`
    - allowed values are `{split, records, index, columns, values, table}`

  The format of the JSON string

<table>
<thead>
<tr>
<th>Format</th>
<th>JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>split</code></td>
<td>dict like <code>{index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</code></td>
</tr>
<tr>
<td><code>records</code></td>
<td>list like <code>[[column -&gt; value], .... , [column -&gt; value]]</code></td>
</tr>
<tr>
<td><code>index</code></td>
<td>dict like <code>{index -&gt; [column -&gt; value]}</code></td>
</tr>
<tr>
<td><code>columns</code></td>
<td>dict like <code>{column -&gt; [index -&gt; value]}</code></td>
</tr>
<tr>
<td><code>values</code></td>
<td>just the values array</td>
</tr>
<tr>
<td><code>table</code></td>
<td>adhering to the JSON Table Schema</td>
</tr>
</tbody>
</table>

- **dtype**: if True, infer dtypes, if a dict of column to dtype, then use those, if False, then don’t infer dtypes at all, default is True, apply only to the data.

- **convert_axes**: boolean, try to convert the axes to the proper dtypes, default is True

- **convert_dates**: a list of columns to parse for dates; If True, then try to parse date-like columns, default is True.

- **keep_default_dates**: boolean, default True. If parsing dates, then parse the default date-like columns.

- **numpy**: direct decoding to NumPy arrays. default is False; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.

- **precise_float**: boolean, default False. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (False) is to use fast but less precise built-in functionality.

2.4. **IO tools (text, CSV, HDF5, …)**
• `date_unit`: string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of ‘s’, ‘ms’, ‘us’ or ‘ns’ to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.

• `lines`: reads file as one json object per line.

• `encoding`: The encoding to use to decode py3 bytes.

• `chunksize`: when used in combination with `lines=True`, return a JsonReader which reads in `chunksize` lines per iteration.

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parseable.

If a non-default `orient` was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see `Orient Options` for an overview.

**Data conversion**

The default of `convert_axes=True`, `dtype=True`, and `convert_dates=True` will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to `dtype`. `convert_axes` should only be set to `False` if you need to preserve string-like numbers (e.g. ‘1’, ‘2’) in an axes.

**Note:** Large integer values may be converted to dates if `convert_dates=True` and the data and / or column labels appear ‘date-like’. The exact threshold depends on the `date_unit` specified. ‘date-like’ means that the column label meets one of the following criteria:

- it ends with `_at`
- it ends with `_time`
- it begins with 'timestamp'
- it is 'modified'
- it is 'date'

**Warning:** When reading JSON data, automatic coercing into dtypes has some quirks:

- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was `float` data will be converted to `integer` if it can be done safely, e.g. a column of 1.
- `bool` columns will be converted to `integer` on reconstruction

Thus there are times where you may want to specify specific dtypes via the `dtype` keyword argument.

Reading from a JSON string:

```
In [232]: pd.read_json(json)
Out[232]:
   date       B     A
0 2013-01-01 2.565646 -1.206412
1 2013-01-01 1.340309  1.431256
2 2013-01-01 -0.226169 -1.170299
3 2013-01-01  0.813850  0.410835
4 2013-01-01 -0.827317  0.132003
```
Reading from a file:

```
In [233]: pd.read_json("test.json")
Out[233]:
A  B  date   ints   bools
2013-01-01 -1.294524 0.413738 2013-01-01 0   True
2013-01-02 0.276662 -0.472035 2013-01-01 1   True
2013-01-03 -0.013960 -0.362543 2013-01-01 2   True
2013-01-04 -0.006154 -0.923061 2013-01-01 3   True
2013-01-05 0.895717 0.805244 2013-01-01 4   True
```

Don’t convert any data (but still convert axes and dates):

```
In [234]: pd.read_json("test.json", dtype=object).dtypes
Out[234]:
A     object
B     object
date object
ints  object
bools object
dtype: object
```

Specify dtypes for conversion:

```
In [235]: pd.read_json("test.json", dtype={"A": "float32", "bools": "int8"}).dtypes
Out[235]:
A     float32
B     float64
date  datetime64[ns]
ints  int64
bools int8
dtype: object
```

Preserve string indices:

```
In [236]: si = pd.DataFrame(
       ...:     np.zeros((4, 4)), columns=list(range(4)), index=[str(i) for i in range(4)]
       ...:     )
       ...

In [237]: si
Out[237]:
   0  1  2  3
0  0.0 0.0 0.0 0.0
1  0.0 0.0 0.0 0.0
2  0.0 0.0 0.0 0.0
3  0.0 0.0 0.0 0.0

In [238]: si.index
Out[238]: Index(['0', '1', '2', '3'], dtype='object')

In [239]: si.columns
Out[239]: Int64Index([0, 1, 2, 3], dtype='int64')

(continues on next page)
In [240]: json = si.to_json()

In [241]: sij = pd.read_json(json, convert_axes=False)

In [242]: sij
Out[242]:
   0  1  2  3
0  0  0  0  0
1  0  0  0  0
2  0  0  0  0
3  0  0  0  0

In [243]: sij.index
Out[243]: Index(['0', '1', '2', '3'], dtype='object')

In [244]: sij.columns
Out[244]: Index(['0', '1', '2', '3'], dtype='object')

Dates written in nanoseconds need to be read back in nanoseconds:

In [245]: json = dfj2.to_json(date_unit="ns")

In [246]: dfju = pd.read_json(json, date_unit="ms")

In [247]: dfju
Out[247]:
   A           B         date        ints   bools
0 -1.294524   0.413738  2013-01-01  False
1  0.276662  -0.472035  2013-01-01  False
2 -0.013960  -0.362543  2013-01-01  False
3 -0.006154  -0.923061  2013-01-01  False
4  0.895717   0.805244  2013-01-01  False

# Let pandas detect the correct precision
In [248]: dfju = pd.read_json(json)

In [249]: dfju
Out[249]:
   A           B         date        ints   bools
0 -1.294524   0.413738  2013-01-01  False
1  0.276662  -0.472035  2013-01-01  False
2 -0.013960  -0.362543  2013-01-01  False
3 -0.006154  -0.923061  2013-01-01  False
4  0.895717   0.805244  2013-01-01  False

# Or specify that all timestamps are in nanoseconds
In [250]: dfju = pd.read_json(json, date_unit="ns")

In [251]: dfju
Out[251]:
   A           B         date        ints   bools
0 -1.294524   0.413738  2013-01-01  False
1  0.276662  -0.472035  2013-01-01  False
2 -0.013960  -0.362543  2013-01-01  False
3 -0.006154  -0.923061  2013-01-01  False
4  0.895717   0.805244  2013-01-01  False
The Numpy parameter

**Note:** This param has been deprecated as of version 1.0.0 and will raise a `FutureWarning`. This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

If `numpy=True` is passed to `read_json` an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```python
In [252]: randfloats = np.random.uniform(-100, 1000, 10000)
In [253]: randfloats.shape = (1000, 10)
In [254]: dffloats = pd.DataFrame(randfloats, columns=list("ABCDEFGHIJ"))
In [255]: jsonfloats = dffloats.to_json()
In [256]: %timeit pd.read_json(jsonfloats)
6.72 ms +- 243 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
In [257]: %timeit pd.read_json(jsonfloats, numpy=True)
4.66 ms +- 117 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```python
In [258]: jsonfloats = dffloats.head(100).to_json()
In [259]: %timeit pd.read_json(jsonfloats)
4.25 ms +- 157 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
In [260]: %timeit pd.read_json(jsonfloats, numpy=True)
3.58 ms +- 14.7 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

**Warning:** Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:
- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A `ValueError` may be raised, or incorrect output may be produced if this condition is not satisfied.
labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using to_json but may not be the case if the JSON is from another source.

Normalization

pandas provides a utility function to take a dict or list of dicts and normalize this semi-structured data into a flat table.

```
In [261]: data = [
    ...:     {"id": 1, "name": {"first": "Coleen", "last": "Volk"}},
    ...:     {"name": {"given": "Mark", "family": "Regner"}},
    ...:     {"id": 2, "name": "Faye Raker"},
    ...: ]

In [262]: pd.json_normalize(data)
Out[262]:
    id  name.first  name.last  name.given  name.family  name
 0   1       Coleen    Volk         NaN          NaN      NaN
 1  NaN       NaN       NaN      Mark         Regner       NaN
 2  2.0       NaN       NaN          NaN            NaN    Faye Raker
```

```
In [263]: data = [
    ...:     {
    ...:         "state": "Florida",
    ...:         "shortname": "FL",
    ...:         "info": {"governor": "Rick Scott"},
    ...:         "county": [
    ...:             {"name": "Dade", "population": 12345},
    ...:             {"name": "Broward", "population": 40000},
    ...:             {"name": "Palm Beach", "population": 60000},
    ...:         ],
    ...: },
    ...: {
    ...:     "state": "Ohio",
    ...:     "shortname": "OH",
    ...:     "info": {"governor": "John Kasich"},
    ...:     "county": [
    ...:         {"name": "Summit", "population": 1234},
    ...:         {"name": "Cuyahoga", "population": 1337},
    ...:     ],
    ...: },
    ...: ]

In [264]: pd.json_normalize(data, "county", ["state", "shortname", ["info", "governor"]])
Out[264]:
     name  population    state   shortname   info.governor
 0  Dade     12345  Florida      FL    Rick Scott
 1  Broward  40000  Florida      FL    Rick Scott
 2 Palm Beach  60000  Florida      FL    Rick Scott
```

(continues on next page)
The max_level parameter provides more control over which level to end normalization. With max_level=1 the following snippet normalizes until 1st nesting level of the provided dict.

```
In [265]: data = [
            ....:     {  
            ....:         "CreatedBy": {"Name": "User001"},  
            ....:         "Lookup": {  
            ....:             "TextField": "Some text",  
            ....:             "UserField": {"Id": "ID001", "Name": "Name001"},  
            ....:         },  
            ....:         "Image": {"a": "b"},  
            ....:     }  
            ....: ]
```

```
In [266]: pd.json_normalize(data, max_level=1)
Out[266]:
CreatedBy.Name Lookup.TextField Lookup.UserField Image.a
0 User001 Some text {"Id": "ID001", "Name": "Name001"} b
```

### Line delimited json

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

For line-delimited json files, pandas can also return an iterator which reads in chunksize lines at a time. This can be useful for large files or to read from a stream.

```
In [267]: jsonl = 
            ....:     "{"a": 1, "b": 2}"
            ....:     "{"a": 3, "b": 4}"
            ....:     ""
```

```
In [268]: df = pd.read_json(jsonl, lines=True)
In [269]: df
Out[269]:
a b
0 1 2
1 3 4
```

```
In [270]: df.to_json(orient="records", lines=True)
Out[270]: '{"a":1,"b":2}
\n{"a":3,"b":4}\n'
```

# reader is an iterator that returns `chunksize` lines each iteration
```
In [271]: with pd.read_json(StringIO(jsonl), lines=True, chunksize=1) as reader:
            ....:     reader
            ....:     for chunk in reader:
```

Table schema

Table Schema is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the orient table to build a JSON string with two fields, schema and data.

```
In [272]: df = pd.DataFrame(
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
    .....:     
)  

In [273]: df
Out[273]:
A B C
idx
0 1 a 2016-01-01
1 2 b 2016-01-02
2 3 c 2016-01-03
```

The schema field contains the fields key, which itself contains a list of column name to type pairs, including the Index or MultiIndex (see below for a list of types). The schema field also contains a primaryKey field if the (Multi)index is unique.

The second field, data, contains the serialized data with the records orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:
A few notes on the generated table schema:

- The `schema` object contains a `pandas_version` field. This contains the version of pandas' dialect of the schema, and will be incremented with each revision.

- All dates are converted to UTC when serializing. Even timezone naive values, which are treated as UTC with an offset of 0.

```python
In [275]: from pandas.io.json import build_table_schema

In [276]: s = pd.Series(pd.date_range("2016", periods=4))

In [277]: build_table_schema(s)
Out[277]:
{
    'fields': [{'name': 'index', 'type': 'integer'},
               {'name': 'values', 'type': 'datetime'}],
    'primaryKey': ['index'],
    'pandas_version': '1.4.0'}
```

- Datetimes with a timezone (before serializing), include an additional field `tz` with the time zone name (e.g. 'US/Central').

```python
In [278]: s_tz = pd.Series(pd.date_range("2016", periods=12, tz="US/Central"))

In [279]: build_table_schema(s_tz)
Out[279]:
{
    'fields': [{'name': 'index', 'type': 'integer'},
               {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
    'primaryKey': ['index'],
    'pandas_version': '1.4.0'}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain an additional field `freq` with the period's frequency, e.g. 'A-DEC'.

```python
In [280]: s_per = pd.Series(1, index=pd.period_range("2016", freq="A-DEC", periods=4))

In [281]: build_table_schema(s_per)
Out[281]:
{
    'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
               {'name': 'values', 'type': 'integer'}],
    'primaryKey': ['index'],
    'pandas_version': '1.4.0'}
```

- Categoricals use the `any` type and an `enum` constraint listing the set of possible values. Additionally, an `ordered` field is included:
In [282]: s_cat = pd.Series(pd.Categorical(["a", "b", "a"]))

In [283]: build_table_schema(s_cat)
Out[283]:
{
    'fields': [
        {
            'name': 'index',
            'type': 'integer',
        },
        {
            'name': 'values',
            'type': 'any',
            'constraints': {
                'enum': ['a', 'b'],
                'ordered': False,
            },
            'primaryKey': ['index'],
        }
    ],
    'primaryKey': ['index'],
    'pandas_version': '1.4.0'
}

• A primary key field, containing an array of labels, is included if the index is unique:

In [284]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [285]: build_table_schema(s_dupe)
Out[285]:
{
    'fields': [
        {
            'name': 'index',
            'type': 'integer',
        },
        {
            'name': 'values',
            'type': 'integer',
        }
    ],
    'pandas_version': '1.4.0'
}

• The primary key behavior is the same with MultiIndex, but in this case the primary key is an array:

In [286]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([("a", "b"), (0, 1)]))

In [287]: build_table_schema(s_multi)
Out[287]:
{
    'fields': [
        {
            'name': 'level_0',
            'type': 'string',
        },
        {
            'name': 'level_1',
            'type': 'integer',
        },
        {
            'name': 'values',
            'type': 'integer',
        }
    ],
    'primaryKey': FrozenList(["level_0", "level_1"]),
    'pandas_version': '1.4.0'
}

• The default naming roughly follows these rules:
  – For series, the object.name is used. If that’s none, then the name is values
  – For DataFrames, the stringified version of the column name is used
  – For Index (not MultiIndex), index.name is used, with a fallback to index if that is None.
  – For MultiIndex, mi.names is used. If any level has no name, then level_<i> is used.

read_json also accepts orient='table' as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner:

In [288]: df = pd.DataFrame(  
        {  
            "foo": [1, 2, 3, 4],  
            "bar": ["a", "b", "c", "d"],  
            "baz": pd.date_range("2018-01-01", freq="d", periods=4),  
            "qux": pd.Categorical(["a", "b", "c", "c"]),  
        },  
        index=pd.Index(range(4), name="idx"),  
    )
In [289]: df
Out[289]:
    foo    bar       baz  qux
idx
0   1    a  2018-01-01 a
1   2    b  2018-01-02 b
2   3    c  2018-01-03 c
3   4    d  2018-01-04 c

In [290]: df.dtypes
Out[290]:
foo    int64
bar    object
baz    datetime64[ns]
qux    category
dtype: object

In [291]: df.to_json("test.json", orient="table")

In [292]: new_df = pd.read_json("test.json", orient="table")

In [293]: new_df
Out[293]:
    foo    bar       baz  qux
idx
0   1    a  2018-01-01 a
1   2    b  2018-01-02 b
2   3    c  2018-01-03 c
3   4    d  2018-01-04 c

In [294]: new_df.dtypes
Out[294]:
foo    int64
bar    object
baz    datetime64[ns]
qux    category
dtype: object

Please note that the literal string ‘index’ as the name of an Index is not round-trippable, nor are any names beginning with 'level_' within a MultiIndex. These are used by default in DataFrame.to_json() to indicate missing values and the subsequent read cannot distinguish the intent.

In [295]: df.index.name = "index"

In [296]: df.to_json("test.json", orient="table")

In [297]: new_df = pd.read_json("test.json", orient="table")

In [298]: print(new_df.index.name)
When using `orient='table'` along with user-defined `ExtensionArray`, the generated schema will contain an additional `extDtype` key in the respective `fields` element. This extra key is not standard but does enable JSON roundtrips for extension types (e.g. `read_json(df.to_json(orient="table"), orient="table")`).

The `extDtype` key carries the name of the extension, if you have properly registered the `ExtensionDtype`, pandas will use said name to perform a lookup into the registry and re-convert the serialized data into your custom dtype.

### 2.4.3 HTML

#### Reading HTML content

**Warning:** We **highly encourage** you to read the *HTML Table Parsing gotchas* below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level `read_html()` function can accept an HTML string/file/URL and will parse HTML tables into list of pandas `DataFrame` objects. Let’s look at a few examples.

**Note:** `read_html` returns a list of `DataFrame` objects, even if there is only a single table contained in the HTML content.

Read a URL with no options:

```python
In [299]: url = "https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list"

In [300]: dfs = pd.read_html(url)

In [301]: dfs
```

```
Out[301]:
```

```
<table>
<thead>
<tr>
<th>Bank Name</th>
<th>Bank City</th>
<th>Bank State</th>
<th>AI</th>
<th>Closing Date</th>
<th>Fund</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquiring InstitutionAI</td>
<td>Almena State Bank</td>
<td>Almena</td>
<td>KS</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Equity Bank</td>
<td>October 23, 2020</td>
<td>10538</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First City Bank of Florida</td>
<td>Fort Walton Beach</td>
<td>FL</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>United Fidelity Bank, fsb</td>
<td>October 16, 2020</td>
<td>10537</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The First State Bank</td>
<td>Barboursville</td>
<td>WV</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>MVB Bank, Inc.</td>
<td>April 3, 2020</td>
<td>10536</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ericson State Bank</td>
<td>Ericson</td>
<td>NE</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Farmers and Merchants Bank</td>
<td>February 14, 2020</td>
<td>10535</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>City National Bank of New Jersey</td>
<td>Newark</td>
<td>NJ</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Industrial Bank</td>
<td>November 1, 2019</td>
<td>10534</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Superior Bank, FSB</td>
<td>Hinsdale</td>
<td>IL</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Superior Federal, FSB</td>
<td>July 27, 2001</td>
<td>6004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Malta National Bank</td>
<td>Malta</td>
<td>OH</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>North Valley Bank</td>
<td>May 3, 2001</td>
<td>4648</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
## Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string:

```python
In [302]: with open(file_path, "r") as f:
    .....
    dfs = pd.read_html(f.read())
    .....

In [303]: dfs
Out[303]:
[ Bank Name City ... Closing Date
 0 Banks of Wisconsin d/b/a Bank of Kenosha Kenosha ... May 31, 2013
 1 May 31, 2013
 2 Central Arizona Bank Scottsdale ... May 14, 2013
 3 May 20, 2013
 4 Sunrise Bank Valdosta ... May 10, 2013
 5 May 21, 2013
 6 Pisgah Community Bank Asheville ... May 10, 2013
 7 May 14, 2013
 8 Douglas County Bank Douglasville ... April 26, 2013
 9 May 16, 2013
 ... ... ... ...
501 Superior Bank, FSB Hinsdale ... July 27, 2001
502 June 5, 2012
503 November 18, 2002
504 First Alliance Bank & Trust Co. Manchester ... February 2, 2001
505 February 18, 2003
504 National State Bank of Metropolis Metropolis ... December 14, 2000
... March 17, 2005
505 Bank of Honolulu Honolulu ... October 13, 2000
505 March 17, 2005
[506 rows x 7 columns]]
```

You can even pass in an instance of `StringIO` if you so desire:

```python
In [304]: with open(file_path, "r") as f:
    .....
    sio = StringIO(f.read())
```

(continues on next page)
In [305]: dfs = pd.read_html(sio)

In [306]: dfs
Out[306]:

<table>
<thead>
<tr>
<th></th>
<th>Bank Name</th>
<th>City</th>
<th>...</th>
<th>Closing Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Banks of Wisconsin d/b/a Bank of Kenosha</td>
<td>Kenosha</td>
<td></td>
<td>May 31, 2013</td>
</tr>
<tr>
<td>1</td>
<td>Central Arizona Bank</td>
<td>Scottsdale</td>
<td></td>
<td>May 14, 2013</td>
</tr>
<tr>
<td>2</td>
<td>Sunrise Bank</td>
<td>Valdosta</td>
<td></td>
<td>May 10, 2013</td>
</tr>
<tr>
<td>3</td>
<td>Pisgah Community Bank</td>
<td>Asheville</td>
<td></td>
<td>May 10, 2013</td>
</tr>
<tr>
<td>4</td>
<td>Douglas County Bank</td>
<td>Douglasville</td>
<td></td>
<td>April 26, 2013</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>501</td>
<td>Superior Bank, FSB</td>
<td>Hinsdale</td>
<td></td>
<td>July 27, 2001</td>
</tr>
<tr>
<td>502</td>
<td>Malta National Bank</td>
<td>Malta</td>
<td></td>
<td>May 3, 2001</td>
</tr>
<tr>
<td>503</td>
<td>First Alliance Bank &amp; Trust Co.</td>
<td>Manchester</td>
<td></td>
<td>February 2, 2001</td>
</tr>
<tr>
<td>504</td>
<td>National State Bank of Metropolis</td>
<td>Metropolis</td>
<td></td>
<td>December 14, 2000</td>
</tr>
<tr>
<td>505</td>
<td>Bank of Honolulu</td>
<td>Honolulu</td>
<td></td>
<td>October 13, 2000</td>
</tr>
</tbody>
</table>

[506 rows x 7 columns]

Note: The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn’t run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text:

```python
match = "Metcalf Bank"
df_list = pd.read_html(url, match=match)
```

Specify a header row (by default `<th>` or `<td>` elements located within a `<thead>` are used to form the column index, if multiple rows are contained within `<thead>` then a MultiIndex is created); if specified, the header row is taken from the data minus the parsed header elements (`<th>` elements).

```python
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```python
dfs = pd.read_html(url, index_col=0)
```
Specify a number of rows to skip:

```python
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (range works as well):

```python
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```python
dfs1 = pd.read_html(url, attrs={"id": "table"})
dfs2 = pd.read_html(url, attrs={"class": "sortable"})
print(np.array_equal(dfs1[0], dfs2[0]))  # Should be True
```

Specify values that should be converted to NaN:

```python
dfs = pd.read_html(url, na_values=["No Acquirer"])
```

Specify whether to keep the default set of NaN values:

```python
dfs = pd.read_html(url, keep_default_na=False)
```

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```python
url_mcc = "https://en.wikipedia.org/wiki/Mobile_country_code"
dfs = pd.read_html(
    url_mcc,
    match="Telekom Albania",
    header=0,
    converters={"MNC": str},
)
```

Use some combination of the above:

```python
dfs = pd.read_html(url, match="Metcalf Bank", index_col=0)
```

Read in pandas to_html output (with some loss of floating point precision):

```python
df = pd.DataFrame(np.random.randn(2, 2))
s = df.to_html(float_format="{0:.40g}".format)
dfin = pd.read_html(s, index_col=0)
```

The lxml backend will raise an error on a failed parse if that is the only parser you provide. If you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings. You may use:

```python
dfs = pd.read_html(url, "Metcalf Bank", index_col=0, flavor="lxml")
```

Or you could pass flavor='lxml' without a list:

```python
dfs = pd.read_html(url, "Metcalf Bank", index_col=0, flavor="lxml")
```

However, if you have bs4 and html5lib installed and pass None or ['lxml', 'bs4'] then the parse will most likely succeed. Note that as soon as a parse succeeds, the function will return.

2.4. IO tools (text, CSV, HDF5, ...) 331
Writing to HTML files

DataFrame objects have an instance method `to_html` which renders the contents of the DataFrame as an HTML table. The function arguments are as in the method `to_string` described above.

**Note:** Not all of the possible options for `DataFrame.to_html` are shown here for brevity’s sake. See `to_html()` for the full set of options.
HTML:

`float_format` takes a Python callable to control the precision of floating point values:

```python
In [311]: print(df.to_html(float_format="{0:.10f}".format))
```

```
</tr>
</thead>
<tbody>
<tr>
    <th>0</th>
    <td>-0.184744</td>
</tr>
<tr>
    <th>1</th>
    <td>-0.856240</td>
</tr>
</tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off:

```python
In [312]: print(df.to_html(bold_rows=False))
```

```
</tr>
</thead>
<tbody>
<tr>
    <th>0</th>
    <td>-0.1847438576</td>
    <td>0.4969711327</td>
</tr>
<tr>
    <th>1</th>
    <td>-0.8562396763</td>
    <td>1.8579766508</td>
</tr>
</tbody>
</table>
```
The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are appended to the existing 'dataframe' class.

```python
In [313]: print(df.to_html(classes=["awesome_table_class", "even_more_awesome_class"]))
<table border="1" class="dataframe awesome_table_class even_more_awesome_class"
  <thead>
  <tr style="text-align: right;">  
    <th></th>
    <th>0</th>
    <th>1</th>
  </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

The `render_links` argument provides the ability to add hyperlinks to cells that contain URLs.

```python
In [314]: url_df = pd.DataFrame(
    .....:     {  
    .....:       "name": ["Python", "pandas"],
    .....:       "url": ["https://www.python.org/", "https://pandas.pydata.org"],
    .....:     }
    .....: )
In [315]: print(url_df.to_html(render_links=True))
```

(continues on next page)
Finally, the escape argument allows you to control whether the “<”, “>” and “&” characters escaped in the resulting HTML (by default it is True). So to get the HTML without escaped characters pass escape=False

```python
In [316]: df = pd.DataFrame({'a': list('&<'), 'b': np.random.randn(3)})
```

Escaped:

```python
In [317]: print(df.to_html(escape=True))
```

(continues on next page)
<tr>
  <th>2</th>
  <td>&gt;</td>
  <td>-0.400654</td>
</tr>
</tbody>
</table>

Not escaped:

```
In [318]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
<thead>
  <tr style="text-align: right;">
    <th></th>
    <th>a</th>
    <th>b</th>
  </tr>
</thead>
<tbody>
<tr>
  <th>0</th>
  <td>&</td>
  <td>-0.474063</td>
</tr>
<tr>
  <th>1</th>
  <td><</td>
  <td>-0.230305</td>
</tr>
<tr>
  <th>2</th>
  <td>></td>
  <td>-0.400654</td>
</tr>
</tbody>
</table>
```

Note: Some browsers may not show a difference in the rendering of the previous two HTML tables.

### HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function `read_html`.

**Issues with lxml**

- **Benefits**
  - `lxml` is very fast.
  - `lxml` requires Cython to install correctly.
• Drawbacks
  – lxml does not make any guarantees about the results of its parse unless it is given strictly valid markup.
  – In light of the above, we have chosen to allow you, the user, to use the lxml backend, but this backend will use html5lib if lxml fails to parse
  – It is therefore highly recommended that you install both BeautifulSoup4 and html5lib, so that you will still get a valid result (provided everything else is valid) even if lxml fails.

Issues with BeautifulSoup4 using lxml as a backend
  • The above issues hold here as well since BeautifulSoup4 is essentially just a wrapper around a parser backend.

Issues with BeautifulSoup4 using html5lib as a backend
  • Benefits
    – html5lib is far more lenient than lxml and consequently deals with real-life markup in a much saner way rather than just, e.g., dropping an element without notifying you.
    – html5lib generates valid HTML5 markup from invalid markup automatically. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is “correct”, since the process of fixing markup does not have a single definition.
    – html5lib is pure Python and requires no additional build steps beyond its own installation.
  • Drawbacks
    – The biggest drawback to using html5lib is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

2.4.4 LaTeX

New in version 1.3.0.
Currently there are no methods to read from LaTeX, only output methods.

Writing to LaTeX files

Note: DataFrame and Styler objects currently have a to_latex method. We recommend using the Styler.to_latex() method over DataFrame.to_latex() due to the former’s greater flexibility with conditional styling, and the latter’s possible future deprecation.

Review the documentation for Styler.to_latex, which gives examples of conditional styling and explains the operation of its keyword arguments.

For simple application the following pattern is sufficient.

```python
In [319]: df = pd.DataFrame([[1, 2], [3, 4]], index=['a', 'b'], columns=['c', 'd'])

In [320]: print(df.style.to_latex())
\begin{tabular}{lrr}
 & c & d \\
a & 1 & 2 \\
\end{tabular}
```

(continues on next page)
To format values before output, chain the `Styler.format` method.

```python
In [321]: print(df.style.format("€ {}") .to_latex())
```

\begin{tabular}{lrr}
\hline
& c & d \\
\hline
a & € 1 & € 2 \\
b & € 3 & € 4 \\
\hline
\end{tabular}

## 2.4.5 XML

### Reading XML

New in version 1.3.0.

The top-level `read_xml()` function can accept an XML string/file/URL and will parse nodes and attributes into a pandas `DataFrame`.

**Note:** Since there is no standard XML structure where design types can vary in many ways, `read_xml` works best with flatter, shallow versions. If an XML document is deeply nested, use the `stylesheet` feature to transform XML into a flatter version.

Let's look at a few examples.

Read an XML string:

```python
In [322]: xml = """"<?xml version="1.0" encoding="UTF-8"?>
........: <bookstore>
........:   <book category="cooking">
........:     <title lang="en">Everyday Italian</title>
........:     <author>Giada De Laurentiis</author>
........:     <year>2005</year>
........:     <price>30.00</price>
........:   </book>
........: <book category="children">
........:   <title lang="en">Harry Potter</title>
........:   <author>J K. Rowling</author>
........:   <year>2005</year>
........:   <price>29.99</price>
........: </book>
........: <book category="web">
........:   <title lang="en">Learning XML</title>
........:   <author>Erik T. Ray</author>
........:   <year>2003</year>
........:   <price>39.95</price>
........: </book>
........: </bookstore>"
```

(continues on next page)
In [323]: df = pd.read_xml(xml)

In [324]: df
Out[324]:
   category    title          author           year    price
0    cooking     Everyday Italian Giada De Laurentiis 2005     30.00
2       web     Learning XML                Erik T. Ray 2003     39.95

Read a URL with no options:

In [325]: df = pd.read_xml("https://www.w3schools.com/xml/books.xml")

In [326]: df
Out[326]:
   category    title          author           year    price    cover
0    cooking     Everyday Italian Giada De Laurentiis 2005     30.00    None
1    children       Harry Potter          J K. Rowling 2005     29.99    None
2       web     XQuery Kick Start Vaidyanathan Nagarajan 2003     49.99    None
3       web     Learning XML                Erik T. Ray 2003     39.95    paperback

Read in the content of the “books.xml” file and pass it to read_xml as a string:

In [327]: with open(file_path, "r") as f:
       ....:     sio = StringIO(f.read())
       ....:
In [328]: df = pd.read_xml(sio)

In [329]: df
Out[329]:
   category    title          author           year    price
0    cooking     Everyday Italian Giada De Laurentiis 2005     30.00
2       web     Learning XML                Erik T. Ray 2003     39.95

Read in the content of the “books.xml” as instance of StringIO or BytesIO and pass it to read_xml:

In [330]: with open(file_path, "rb") as f:
       ....:     bio = BytesIO(f.read())

In [331]: df = pd.read_xml(bio)

In [332]: df
Out[332]:
   category    title          author           year    price
0    cooking     Everyday Italian Giada De Laurentiis 2005     30.00
2       web     Learning XML                Erik T. Ray 2003     39.95

(continues on next page)
In [333]: df = pd.read_xml(bio)

In [334]: df
Out[334]:
<table>
<thead>
<tr>
<th>category</th>
<th>title</th>
<th>author</th>
<th>year</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooking</td>
<td>Everyday Italian</td>
<td>Giada De Laurentiis</td>
<td>2005</td>
<td>30.00</td>
</tr>
<tr>
<td>web</td>
<td>Learning XML</td>
<td>Erik T. Ray</td>
<td>2003</td>
<td>39.95</td>
</tr>
</tbody>
</table>

Even read XML from AWS S3 buckets such as Python Software Foundation’s IRS 990 Form:

In [335]: df = pd.read_xml(
               "s3://irs-form-990/201923199349319487_public.xml",
               xpath="//irs:Form990PartVIISectionAGrp",
               namespaces={"irs": "http://www.irs.gov/efile"}
)

In [336]: df
Out[336]:
<table>
<thead>
<tr>
<th>PersonNm</th>
<th>TitleTxt</th>
<th>OtherCompensationAmt</th>
<th>HighestCompensatedEmployeeInd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naomi Ceder</td>
<td>Chair</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Van Lindberg</td>
<td>Vice Chair, General Counsel</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Kurt Kaiser</td>
<td>Treasurer</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Ewa Jodlowska</td>
<td>Secretary, Director of Operations</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Thomas Wouters</td>
<td>Director, Vice Chair</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Kenneth Reitz</td>
<td>Director</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Jeffrey Triplett</td>
<td>Director</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Betsy Waliszewski</td>
<td>Assistant Secretary, Event Coordinator</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Guido van Rossum</td>
<td>President</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Ernest W Durbin III</td>
<td>Director of Infrastructure, PyCon Chair</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

[21 rows x 10 columns]

With `lxml` as default parser, you access the full-featured XML library that extends Python’s ElementTree API. One powerful tool is ability to query nodes selectively or conditionally with more expressive XPath:
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [337]: df = pd.read_xml(file_path, xpath="//book[year=2005]")
In [338]: df
Out[338]:
category
title
0
cooking Everyday Italian
1 children
Harry Potter

author
Giada De Laurentiis
J K. Rowling

year
2005
2005

price
30.00
29.99

Specify only elements or only attributes to parse:
In [339]: df = pd.read_xml(file_path, elems_only=True)
In [340]: df
Out[340]:
0
1
2

title
Everyday Italian
Harry Potter
Learning XML

author
Giada De Laurentiis
J K. Rowling
Erik T. Ray

year
2005
2005
2003

price
30.00
29.99
39.95

In [341]: df = pd.read_xml(file_path, attrs_only=True)
In [342]: df
Out[342]:
category
0
cooking
1 children
2
web
XML documents can have namespaces with prefixes and default namespaces without prefixes both of which are denoted
with a special attribute xmlns. In order to parse by node under a namespace context, xpath must reference a prefix.
For example, below XML contains a namespace with prefix, doc, and URI at https://example.com. In order to
parse doc:row nodes, namespaces must be used.
In [343]: xml = """<?xml version='1.0' encoding='utf-8'?>
.....: <doc:data xmlns:doc="https://example.com">
.....:
<doc:row>
.....:
<doc:shape>square</doc:shape>
.....:
<doc:degrees>360</doc:degrees>
.....:
<doc:sides>4.0</doc:sides>
.....:
</doc:row>
.....:
<doc:row>
.....:
<doc:shape>circle</doc:shape>
.....:
<doc:degrees>360</doc:degrees>
.....:
<doc:sides/>
.....:
</doc:row>
.....:
<doc:row>
.....:
<doc:shape>triangle</doc:shape>
.....:
<doc:degrees>180</doc:degrees>
.....:
<doc:sides>3.0</doc:sides>
.....:
</doc:row>
.....: </doc:data>"""
.....:
(continues on next page)

2.4. IO tools (text, CSV, HDF5, . . . )

341


In [344]: df = pd.read_xml(xml,
       ..........: xpath="/doc:row",
       ..........: namespaces={"doc": "https://example.com"})

In [345]: df
Out[345]:
shape  degrees  sides
0  square    360     4.0
1   circle   360   NaN
2  triangle  180     3.0

Similarly, an XML document can have a default namespace without prefix. Failing to assign a temporary prefix will return no nodes and raise a ValueError. But assigning any temporary name to correct URI allows parsing by nodes.

In [346]: xml = """<?xml version='1.0' encoding='utf-8'?>
       ..........: <data xmlns="https://example.com">
       ..........:  <row>
       ..........:    <shape>square</shape>
       ..........:    <degrees>360</degrees>
       ..........:    <sides>4.0</sides>
       ..........:  </row>
       ..........:  <row>
       ..........:    <shape>circle</shape>
       ..........:    <degrees>360</degrees>
       ..........:    <sides/>
       ..........:  </row>
       ..........:  <row>
       ..........:    <shape>triangle</shape>
       ..........:    <degrees>180</degrees>
       ..........:    <sides>3.0</sides>
       ..........: </row>
       ..........: </data>""

In [347]: df = pd.read_xml(xml,
       ..........: xpath="/pandas:row",
       ..........: namespaces={"pandas": "https://example.com"})

In [348]: df
Out[348]:
shape  degrees  sides
0  square    360     4.0
1   circle   360   NaN
2  triangle  180     3.0

However, if XPath does not reference node names such as default, /*, then namespaces is not required.

With lxml as parser, you can flatten nested XML documents with an XSLT script which also can be string/file/URL types. As background, XSLT is a special-purpose language written in a special XML file that can transform original XML documents into other XML, HTML, even text (CSV, JSON, etc.) using an XSLT processor.
For example, consider this somewhat nested structure of Chicago “L” Rides where station and rides elements encapsulate data in their own sections. With below XSLT, lxml can transform original nested document into a flatter output (as shown below for demonstration) for easier parse into DataFrame:

```python
In [349]: xml = """<?xml version='1.0' encoding='utf-8'?>
   ....: <response>
   ....:   <row>
   ....:     <station id="40850" name="Library"/>
   ....:     <month>2020-09-01T00:00:00</month>
   ....:     <rides>
   ....:       <avg_weekday_rides>864.2</avg_weekday_rides>
   ....:       <avg_saturday_rides>534</avg_saturday_rides>
   ....:       <avg_sunday_holiday_rides>417.2</avg_sunday_holiday_rides>
   ....:     </rides>
   ....:   </row>
   ....:   <row>
   ....:     <station id="41700" name="Washington/Wabash"/>
   ....:     <month>2020-09-01T00:00:00</month>
   ....:     <rides>
   ....:       <avg_weekday_rides>2707.4</avg_weekday_rides>
   ....:       <avg_saturday_rides>1909.8</avg_saturday_rides>
   ....:       <avg_sunday_holiday_rides>1438.6</avg_sunday_holiday_rides>
   ....:     </rides>
   ....:   </row>
   ....:   <row>
   ....:     <station id="40380" name="Clark/Lake"/>
   ....:     <month>2020-09-01T00:00:00</month>
   ....:     <rides>
   ....:       <avg_weekday_rides>2949.6</avg_weekday_rides>
   ....:       <avg_saturday_rides>1657</avg_saturday_rides>
   ....:       <avg_sunday_holiday_rides>1453.8</avg_sunday_holiday_rides>
   ....:     </rides>
   ....:   </row>
   ....: </response>"
```

```python
In [350]: xsl = """"""""xml" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   ....: <xsl:output method="xml" omit-xml-declaration="no" indent="yes"/>
   ....: <xsl:strip-space elements="*"/>
   ....: <xsl:template match="/response">
   ....:   <xsl:copy>
   ....:     <xsl:apply-templates select="row"/>
   ....:   </xsl:copy>
   ....: </xsl:template>
   ....: <xsl:template match="row">
   ....:   <xsl:copy>
   ....:     <station_id><xsl:value-of select="station/@id"/></station_id>
   ....:     <station_name><xsl:value-of select="station/@name"/></station_name>
   ....:     <month><xsl:value-of select="month|rides/*"/></month>
   ....:   </xsl:copy>
   ....: </xsl:template>
   ....: </xsl:stylesheet>"
```

(continues on next page)
In [351]: output = """<?xml version='1.0' encoding='utf-8'?>
.....: <response>
.....: <row>
.....:   <station_id>40850</station_id>
.....:   <station_name>Library</station_name>
.....:   <month>2020-09-01T00:00:00</month>
.....:   <avg_weekday_rides>864.2</avg_weekday_rides>
.....:   <avg_saturday_rides>534</avg_saturday_rides>
.....:   <avg_sunday_holiday_rides>417.2</avg_sunday_holiday_rides>
.....: </row>
.....: <row>
.....:   <station_id>41700</station_id>
.....:   <station_name>Washington/Wabash</station_name>
.....:   <month>2020-09-01T00:00:00</month>
.....:   <avg_weekday_rides>2707.4</avg_weekday_rides>
.....:   <avg_saturday_rides>1909.8</avg_saturday_rides>
.....:   <avg_sunday_holiday_rides>1438.6</avg_sunday_holiday_rides>
.....: </row>
.....: <row>
.....:   <station_id>40380</station_id>
.....:   <station_name>Clark/Lake</station_name>
.....:   <month>2020-09-01T00:00:00</month>
.....:   <avg_weekday_rides>2949.6</avg_weekday_rides>
.....:   <avg_saturday_rides>1657</avg_saturday_rides>
.....:   <avg_sunday_holiday_rides>1453.8</avg_sunday_holiday_rides>
.....: </row>
.....:
"""

In [352]: df = pd.read_xml(xml, stylesheet=xsl)

In [353]: df
Out[353]:
    station_id  station_name                  month avg_weekday_rides  avg_saturday_rides avg_sunday_holiday_rides
0        40850          Library 2020-09-01T00:00:00      864.2          534.0                417.2
1        41700  Washington/Wabash 2020-09-01T00:00:00      2707.4         1909.8               1438.6
2        40380          Clark/Lake 2020-09-01T00:00:00     2949.6          1657.0               1453.8
Writing XML

New in version 1.3.0.

**DataFrame** objects have an instance method `to_xml` which renders the contents of the DataFrame as an XML document.

**Note:** This method does not support special properties of XML including DTD, CDATA, XSD schemas, processing instructions, comments, and others. Only namespaces at the root level is supported. However, stylesheet allows design changes after initial output.

Let's look at a few examples.

Write an XML without options:

```python
In [354]: geom_df = pd.DataFrame(
    .....:     {  
    .....:         "shape": ["square", "circle", "triangle"],  
    .....:         "degrees": [360, 360, 180],   
    .....:         "sides": [4, np.nan, 3],     
    .....:     }  
    .....: )  
```

```python
In [355]: print(geom_df.to_xml())
<?xml version='1.0' encoding='utf-8'>
<data>
    <row>
        <index>0</index>
        <shape>square</shape>
        <degrees>360</degrees>
        <sides>4.0</sides>
    </row>
    <row>
        <index>1</index>
        <shape>circle</shape>
        <degrees>360</degrees>
        <sides/>
    </row>
    <row>
        <index>2</index>
        <shape>triangle</shape>
        <degrees>180</degrees>
        <sides>3.0</sides>
    </row>
</data>
```

Write an XML with new root and row name:

```python
In [356]: print(geom_df.to_xml(root_name="geometry", row_name="objects"))
<?xml version='1.0' encoding='utf-8'>
<geometry>
    <objects>
        <row>
            <index>0</index>
            <shape>square</shape>
            <degrees>360</degrees>
            <sides>4.0</sides>
        </row>
        <row>
            <index>1</index>
            <shape>circle</shape>
            <degrees>360</degrees>
            <sides/>
        </row>
        <row>
            <index>2</index>
            <shape>triangle</shape>
            <degrees>180</degrees>
            <sides>3.0</sides>
        </row>
    </objects>
</geometry>
```

(continues on next page)
Write an attribute-centric XML:

```
In [357]: print(geom_df.to_xml(attr_cols=geom_df.columns.tolist()))
<?xml version='1.0' encoding='utf-8'?><data>
    <row index="0" shape="square" degrees="360" sides="4.0"/>
    <row index="1" shape="circle" degrees="360"/>
    <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

Write a mix of elements and attributes:

```
In [358]: print(geom_df.to_xml(
        ....:     index=False,
        ....:     attr_cols=['shape'],
        ....:     elem_cols=['degrees', 'sides']))
<?xml version='1.0' encoding='utf-8'?><data>
    <row shape="square">
        <degrees>360</degrees>
        <sides>4.0</sides>
    </row>
    <row shape="circle">
        <degrees>360</degrees>
        <sides/>
    </row>
    <row shape="triangle">
        <degrees>180</degrees>
        <sides>3.0</sides>
    </row>
</data>
```
Any **DataFrames** with hierarchical columns will be flattened for XML element names with levels delimited by underscores:

```
In [359]: ext_geom_df = pd.DataFrame(
       ...:     {
       ...:         "type": ["polygon", "other", "polygon"],
       ...:         "shape": ["square", "circle", "triangle"],
       ...:         "degrees": [360, 360, 180],
       ...:         "sides": [4, np.nan, 3],
       ...:     }
       ...: )

In [360]: pvt_df = ext_geom_df.pivot_table(index='shape',
       ...:     columns='type',
       ...:     values=['degrees', 'sides'],
       ...:     aggfunc='sum')
```

```
In [361]: pvt_df
Out[361]:
          degrees     sides
   type  other  polygon  other  polygon
shape
circle 360.0      NaN   0.0      NaN
square NaN  360.0    NaN   4.0
triangle NaN  180.0   NaN   3.0
```

```
In [362]: print(pvt_df.to_xml())
<?xml version='1.0' encoding='utf-8'?>
<data>
   <row>
      <shape>circle</shape>
      <degrees_other>360.0</degrees_other>
      <degrees_polygon/>
      <sides_other>0.0</sides_other>
      <sides_polygon/>
   </row>
   <row>
      <shape>square</shape>
      <degrees_other/>
      <degrees_polygon>360.0</degrees_polygon>
      <sides_other/>
      <sides_polygon>4.0</sides_polygon>
   </row>
   <row>
      <shape>triangle</shape>
      <degrees_other/>
      <degrees_polygon>180.0</degrees_polygon>
      <sides_other/>
      <sides_polygon>3.0</sides_polygon>
   </row>
</data>
```
Write an XML with default namespace:

```python
In [363]: print(geom_df.to_xml(namespaces={": "https://example.com"}))
<?xml version='1.0' encoding='utf-8'?>
<data xmlns="https://example.com">
   <row>
      <index>0</index>
      <shape>square</shape>
      <degrees>360</degrees>
      <sides>4.0</sides>
   </row>
   <row>
      <index>1</index>
      <shape>circle</shape>
      <degrees>360</degrees>
      <sides/>
   </row>
   <row>
      <index>2</index>
      <shape>triangle</shape>
      <degrees>180</degrees>
      <sides>3.0</sides>
   </row>
</data>
```

Write an XML with namespace prefix:

```python
In [364]: print(
       ......: geom_df.to_xml(namespaces={"doc": "https://example.com"},
       ......: prefix="doc")
       ......: )
```

```xml
<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
   <doc:row>
      <doc:index>0</doc:index>
      <doc:shape>square</doc:shape>
      <doc:degrees>360</doc:degrees>
      <doc:sides>4.0</doc:sides>
   </doc:row>
   <doc:row>
      <doc:index>1</doc:index>
      <doc:shape>circle</doc:shape>
      <doc:degrees>360</doc:degrees>
      <doc:sides/>
   </doc:row>
   <doc:row>
      <doc:index>2</doc:index>
      <doc:shape>triangle</doc:shape>
      <doc:degrees>180</doc:degrees>
      <doc:sides>3.0</doc:sides>
   </doc:row>
</doc:data>
```
Write an XML without declaration or pretty print:

```python
In [365]:
    print(geom_df.to_xml(xml_declaration=False,
                         pretty_print=False))
```

```xml
<data>
    <row index="0" shape="square" degrees="360" sides="4.0"/>
    <row index="1" shape="circle" degrees="360"/>
    <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

Write an XML and transform with stylesheet:

```python
In [366]:
xsl = """
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output method="xml" omit-xm
declaration="no" indent="yes"/>
        <xsl:strip-space elements="*"/>
        <xsl:template match="/data">
            <geometry>
                <xsl:apply-templates select="row"/>
            </geometry>
        </xsl:template>
        <xsl:template match="row">
            <object index="{index}" type="polygon">
                <shape>{shape}</shape>
                <property>
                    <degrees>{degrees}</degrees>
                    <sides>{sides}</sides>
                </property>
            </object>
        </xsl:template>
    </xsl:stylesheet>"

In [367]:
    print(geom_df.to_xml(stylesheet=xsl))
```

```xml
<?xml version="1.0"?>
<geometry>
    <object index="0" type="polygon">
        <shape>square</shape>
        <property>
            <degrees>360</degrees>
            <sides>4.0</sides>
        </property>
    </object>
    <object index="1">
        <shape>circle</shape>
        <property>
            <degrees>360</degrees>
        </property>
    </object>
    <object index="2">
        <shape>triangle</shape>
        <property>
            <degrees>180</degrees>
            <sides>3.0</sides>
        </property>
    </object>
</geometry>
```

(continues on next page)
XML Final Notes

- All XML documents adhere to [W3C specifications](https://www.w3.org). Both **etree** and **lxml** parsers will fail to parse any markup document that is not well-formed or follows XML syntax rules. Do be aware HTML is not an XML document unless it follows XHTML specs. However, other popular markup types including KML, XAML, RSS, MusicML, MathML are compliant XML schemas.

- For above reason, if your application builds XML prior to pandas operations, use appropriate DOM libraries like **etree** and **lxml** to build the necessary document and not by string concatenation or regex adjustments. Always remember XML is a special text file with markup rules.

- With very large XML files (several hundred MBs to GBs), XPath and XSLT can become memory-intensive operations. Be sure to have enough available RAM for reading and writing to large XML files (roughly about 5 times the size of text).

- Because XSLT is a programming language, use it with caution since such scripts can pose a security risk in your environment and can run large or infinite recursive operations. Always test scripts on small fragments before full run.

- The **etree** parser supports all functionality of both **read_xml** and **to_xml** except for complex XPath and any XSLT. Though limited in features, **etree** is still a reliable and capable parser and tree builder. Its performance may trail **lxml** to a certain degree for larger files but relatively unnoticeable on small to medium size files.

### 2.4.6 Excel files

The **read_excel()** method can read Excel 2007+ (.xlsx) files using the [openpyxl](https://openpyxl.readthedocs.io) Python module. Excel 2003 (.xls) files can be read using **xlrd**. Binary Excel (.xlsb) files can be read using **pyxlsb**. The **to_excel()** instance method is used for saving a **DataFrame** to Excel. Generally the semantics are similar to working with **csv** data. See the [cookbook](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) for some advanced strategies.

**Warning:** The **xlwt** package for writing old-style .xls excel files is no longer maintained. The **xlrd** package is now only for reading old-style .xls files.

Before pandas 1.3.0, the default argument **engine=None** to **read_excel()** would result in using the **xlrd** engine in many cases, including new Excel 2007+ (.xlsx) files. pandas will now default to using the **openpyxl** engine.

It is strongly encouraged to install **openpyxl** to read Excel 2007+ (.xlsx) files. **Please do not report issues when using **xlrd** to read **xlsx** files.** This is no longer supported, switch to using **openpyxl** instead.

Attempting to use the **xlwt** engine will raise a [FutureWarning](https://pandas.pydata.org/pandas-docs/stable/user_guide/future_warnings.html) unless the option `io.excel.xls.writer` is set to "xlwt". While this option is now deprecated and will also raise a FutureWarning, it can be globally set and the warning suppressed. Users are recommended to write .xlsx files using the **openpyxl** engine instead.
Reading Excel files

In the most basic use-case, `read_excel` takes a path to an Excel file, and the `sheet_name` indicating which sheet to parse.

```python
# Returns a DataFrame
pd.read_excel("path_to_file.xls", sheet_name="Sheet1")
```

**ExcelFile class**

To facilitate working with multiple sheets from the same file, the `ExcelFile` class can be used to wrap the file and can be passed into `read_excel` There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```python
xlsx = pd.ExcelFile("path_to_file.xls")
df = pd.read_excel(xlsx, "Sheet1")
```

The `ExcelFile` class can also be used as a context manager.

```python
with pd.ExcelFile("path_to_file.xls") as xls:
    df1 = pd.read_excel(xls, "Sheet1")
    df2 = pd.read_excel(xls, "Sheet2")
```

The `sheet_names` property will generate a list of the sheet names in the file.

The primary use-case for an `ExcelFile` is parsing multiple sheets with different parameters:

```python
data = {}
# For when Sheet1's format differs from Sheet2
with pd.ExcelFile("path_to_file.xls") as xls:
    data["Sheet1"] = pd.read_excel(xls, "Sheet1", index_col=None, na_values=["NA"])
    data["Sheet2"] = pd.read_excel(xls, "Sheet2", index_col=1)
```

Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to `read_excel` with no loss in performance.

```python
# using the ExcelFile class
data = {}
with pd.ExcelFile("path_to_file.xls") as xls:
    data["Sheet1"] = pd.read_excel(xls, "Sheet1", index_col=None, na_values=["NA"])
    data["Sheet2"] = pd.read_excel(xls, "Sheet2", index_col=None, na_values=["NA"])
# equivalent using the read_excel function
data = pd.read_excel(
    "path_to_file.xls", ["Sheet1", "Sheet2"], index_col=None, na_values=["NA"])
```

ExcelFile can also be called with a `xlrd.book.Book` object as a parameter. This allows the user to control how the excel file is read. For example, sheets can be loaded on demand by calling `xlrd.open_workbook()` with `on_demand=True`.

```python
import xlrd
```

(continues on next page)
Specifying sheets

**Note:** The second argument is `sheet_name`, not to be confused with `ExcelFile.sheet_names`.

**Note:** An ExcelFile’s attribute `sheet_names` provides access to a list of sheets.

- The arguments `sheet_name` allows specifying the sheet or sheets to read.
- The default value for `sheet_name` is 0, indicating to read the first sheet.
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.
- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Pass a `None` to return a dictionary of all available sheets.

```python
# Returns a DataFrame
pd.read_excel("path_to_file.xls", "Sheet1", index_col=None, na_values=["NA"])
```

Using the sheet index:

```python
# Returns a DataFrame
pd.read_excel("path_to_file.xls", 0, index_col=None, na_values=["NA"])
```

Using all default values:

```python
# Returns a DataFrame
pd.read_excel("path_to_file.xls")
```

Using `None` to get all sheets:

```python
# Returns a dictionary of DataFrames
pd.read_excel("path_to_file.xls", sheet_name=None)
```

Using a list to get multiple sheets:

```python
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel("path_to_file.xls", sheet_name=["Sheet1", 3])
```

`read_excel` can read more than one sheet, by setting `sheet_name` to either a list of sheet names, a list of sheet positions, or `None` to read all sheets. Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.
Reading a MultiIndex

read_excel can read a MultiIndex index, by passing a list of columns to index_col and a MultiIndex column by passing a list of rows to header. If either the index or columns have serialized level names those will be read in as well by specifying the rows/columns that make up the levels.

For example, to read in a MultiIndex index without names:

```python
In [368]: df = pd.DataFrame(  
      ...:     {"a": [1, 2, 3, 4], "b": [5, 6, 7, 8]},  
      ...:     index=pd.MultiIndex.from_product(["a", "b"], ["c", "d"])),  
      ...: )

In [369]: df.to_excel("path_to_file.xlsx")

In [370]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1])

In [371]: df
Out[371]:
   a  b
a c  1  5
   d  2  6
b c  3  7
   d  4  8
```

If the index has level names, they will parsed as well, using the same parameters.

```python
In [372]: df.index = df.index.set_names(["lvl1", "lvl2"])

In [373]: df.to_excel("path_to_file.xlsx")

In [374]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1])

In [375]: df
Out[375]:
   a  b
1lvl1 lvl2  
   a  c  1  5
   d  2  6
   b  c  3  7
   d  4  8
```

If the source file has both MultiIndex index and columns, lists specifying each should be passed to index_col and header:

```python
In [376]: df.columns = pd.MultiIndex.from_product(["a", ["b", "d"], names=["c1", "c2 ...")

In [377]: df.to_excel("path_to_file.xlsx")

In [378]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1], header=[0, 1])

In [379]: df
```

(continues on next page)
Parsing specific columns

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. read_excel takes a usecols keyword to allow you to specify a subset of columns to parse.

Changed in version 1.0.0.

Passing in an integer for usecols will no longer work. Please pass in a list of ints from 0 to usecols inclusive instead.

You can specify a comma-delimited set of Excel columns and ranges as a string:

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols="A,C:E")
```

If usecols is a list of integers, then it is assumed to be the file column indices to be parsed.

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=[0, 2, 3])
```

Element order is ignored, so usecols=[0, 1] is the same as [1, 0].

If usecols is a list of strings, it is assumed that each string corresponds to a column name provided either by the user in names or inferred from the document header row(s). Those strings define which columns will be parsed:

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=["foo", "bar"])
```

Element order is ignored, so usecols=['baz', 'joe'] is the same as ['joe', 'baz'].

If usecols is callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True.

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=lambda x: x.isalpha())
```

Parsing dates

Datetime-like values are normally automatically converted to the appropriate dtype when reading the excel file. But if you have a column of strings that look like dates (but are not actually formatted as dates in excel), you can use the parse_dates keyword to parse those strings to datetimes:

```
pd.read_excel("path_to_file.xls", "Sheet1", parse_dates=["date_strings"])
```
Cell converters

It is possible to transform the contents of Excel cells via the converters option. For instance, to convert a column to boolean:

```python
pd.read_excel("path_to_file.xls", "Sheet1", converters={"MyBools": bool})
```

This option handles missing values and treats exceptions in the converters as missing data. Transformations are applied cell by cell rather than to the column as a whole, so the array dtype is not guaranteed. For instance, a column of integers with missing values cannot be transformed to an array with integer dtype, because NaN is strictly a float. You can manually mask missing data to recover integer dtype:

```python
def cfun(x):
    return int(x) if x else -1

pd.read_excel("path_to_file.xls", "Sheet1", converters={"MyInts": cfun})
```

Dtype specifications

As an alternative to converters, the type for an entire column can be specified using the dtype keyword, which takes a dictionary mapping column names to types. To interpret data with no type inference, use the type str or object.

```python
pd.read_excel("path_to_file.xls", dtype={"MyInts": "int64", "MyText": str})
```

Writing Excel files

Writing Excel files to disk

To write a DataFrame object to a sheet of an Excel file, you can use the to_excel instance method. The arguments are largely the same as to_csv described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the DataFrame should be written. For example:

```python
df.to_excel("path_to_file.xlsx", sheet_name="Sheet1")
```

Files with a .xls extension will be written using xlwt and those with a .xlsx extension will be written using xlsxwriter (if available) or openpyxl.

The DataFrame will be written in a way that tries to mimic the REPL output. The index_label will be placed in the second row instead of the first. You can place it in the first row by setting the merge_cells option in to_excel() to False:

```python
df.to_excel("path_to_file.xlsx", index_label="label", merge_cells=False)
```

In order to write separate DataFrames to separate sheets in a single Excel file, one can pass an ExcelWriter.

```python
with pd.ExcelWriter("path_to_file.xlsx") as writer:
    df1.to_excel(writer, sheet_name="Sheet1")
    df2.to_excel(writer, sheet_name="Sheet2")
```
**Writing Excel files to memory**

pandas supports writing Excel files to buffer-like objects such as `StringIO` or `BytesIO` using `ExcelWriter`.

```python
from io import BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter(bio, engine="xlsxwriter")
df.to_excel(writer, sheet_name="Sheet1")

# Save the workbook
writer.save()

# Seek to the beginning and read to copy the workbook to a variable in memory
bio.seek(0)
workbook = bio.read()
```

**Note:** engine is optional but recommended. Setting the engine determines the version of workbook produced. Setting `engine='xlrd'` will produce an Excel 2003-format workbook (xls). Using either 'openpyxl' or 'xlsxwriter' will produce an Excel 2007-format workbook (xlsx). If omitted, an Excel 2007-formatted workbook is produced.

**Excel writer engines**

Deprecated since version 1.2.0: As the xlwt package is no longer maintained, the xlwt engine will be removed from a future version of pandas. This is the only engine in pandas that supports writing to `.xls` files.

pandas chooses an Excel writer via two methods:

1. the engine keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the XlsxWriter for `.xlsx`, openpyxl for `.xlsm`, and xlwt for `.xls` files. If you have multiple engines installed, you can set the default engine through setting the config options `io.excel.xlsx.writer` and `io.excel.xls.writer`. pandas will fall back on openpyxl for `.xlsx` files if Xlsxwriter is not available.

To specify which writer you want to use, you can pass an engine keyword argument to `to_excel` and to `ExcelWriter`. The built-in engines are:

- openpyxl: version 2.4 or higher is required
- xlsxwriter
- xlwt

```python
# By setting the 'engine' in the DataFrame 'to_excel()' methods.
df.to_excel("path_to_file.xlsx", sheet_name="Sheet1", engine="xlsxwriter")

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter("path_to_file.xlsx", engine="xlsxwriter")

# Or via pandas configuration.
```

(continues on next page)
from pandas import options  # noqa: E402
options.io.excel.xlsx.writer = "xlsxwriter"
df.to_excel("path_to_file.xlsx", sheet_name="Sheet1")

**Style and formatting**

The look and feel of Excel worksheets created from pandas can be modified using the following parameters on the DataFrame’s `to_excel` method.

- `float_format`: Format string for floating point numbers (default `None`).
- `freeze_panes`: A tuple of two integers representing the bottommost row and rightmost column to freeze. Each of these parameters is one-based, so (1, 1) will freeze the first row and first column (default `None`).

Using the `Xlsxwriter` engine provides many options for controlling the format of an Excel worksheet created with the `to_excel` method. Excellent examples can be found in the `Xlsxwriter` documentation here: https://xlsxwriter.readthedocs.io/working_with_pandas.html

### 2.4.7 OpenDocument Spreadsheets

New in version 0.25.

The `read_excel()` method can also read OpenDocument spreadsheets using the `odfpy` module. The semantics and features for reading OpenDocument spreadsheets match what can be done for Excel files using `engine='odf'`.

```python
# Returns a DataFrame
pd.read_excel("path_to_file.ods", engine="odf")
```

**Note:** Currently pandas only supports reading OpenDocument spreadsheets. Writing is not implemented.

### 2.4.8 Binary Excel (.xlsb) files

New in version 1.0.0.

The `read_excel()` method can also read binary Excel files using the `pyxlsb` module. The semantics and features for reading binary Excel files mostly match what can be done for Excel files using `engine='pyxlsb'`. `pyxlsb` does not recognize datetime types in files and will return floats instead.

```python
# Returns a DataFrame
pd.read_excel("path_to_file.xlsb", engine="pyxlsb")
```

**Note:** Currently pandas only supports reading binary Excel files. Writing is not implemented.
2.4.9 Clipboard

A handy way to grab data is to use the `read_clipboard()` method, which takes the contents of the clipboard buffer and passes them to the `read_csv` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
A  B  C
x 1  4  p
y 2  5  q
z 3  6  r
```

And then import the data directly to a DataFrame by calling:

```python
>>> clipdf = pd.read_clipboard()
>>> clipdf
   A  B  C
0  x 1  4  p
1  y 2  5  q
2  z 3  6  r
```

The `to_clipboard` method can be used to write the contents of a DataFrame to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a DataFrame into clipboard and reading it back.

```python
>>> df = pd.DataFrame(
...     {"A": [1, 2, 3], "B": [4, 5, 6], "C": ["p", "q", "r"],
...     index=["x", "y", "z"]
... )

>>> df
   A  B  C
0  x 1  4  p
1  y 2  5  q
2  z 3  6  r
>>> df.to_clipboard()
>>> pd.read_clipboard()
   A  B  C
0  x 1  4  p
1  y 2  5  q
2  z 3  6  r
```

We can see that we got the same content back, which we had earlier written to the clipboard.

**Note:** You may need to install xclip or xsel (with PyQt5, PyQt4 or qtpy) on Linux to use these methods.
2.4.10 Pickling

All pandas objects are equipped with to_pickle methods which use Python’s cPickle module to save data structures to disk using the pickle format.

```
In [380]: df
c1  a
   c2  b  d
lv1 lv2
   a  c  1  5
       d  2  6
   b  c  3  7
       d  4  8
Out[380]:
```

```
In [381]: df.to_pickle("foo.pkl")
```

The read_pickle function in the pandas namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [382]: pd.read_pickle("foo.pkl")
Out[382]:
```

```
Warning: Loading pickled data received from untrusted sources can be unsafe.
See: https://docs.python.org/3/library/pickle.html
```

```
Warning: read_pickle() is only guaranteed backwards compatible back to pandas version 0.20.3
```

Compressed pickle files

read_pickle(), DataFrame.to_pickle() and Series.to_pickle() can read and write compressed pickle files. The compression types of gzip, bz2, xz, zstd are supported for reading and writing. The zip file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If `infer`, then use gzip, bz2, zip, xz, zstd if filename ends in `.gz`, `.bz2`, `.zip`, `.xz`, or `.zst`, respectively.

The compression parameter can also be a dict in order to pass options to the compression protocol. It must have a 'method' key set to the name of the compression protocol, which must be one of ['zip', 'gzip', 'bz2', 'xz', 'zstd']. All other key-value pairs are passed to the underlying compression library.

```
In [383]: df = pd.DataFrame(
    ....:     }
```
Pandas: powerful Python data analysis toolkit, Release 1.4.2

In [384]: df
Out[384]:
   A     B     C
0 -0.28  foo  2013-01-01 00:00:00
1 -0.08  foo  2013-01-01 00:00:01
2  0.00  foo  2013-01-01 00:00:02
3  1.38  foo  2013-01-01 00:00:03
4  0.34  foo  2013-01-01 00:00:04
..   ...   ...   ...
995 -0.22  foo  2013-01-01 00:16:35
996  0.49  foo  2013-01-01 00:16:36
997 -0.46  foo  2013-01-01 00:16:37
998  1.36  foo  2013-01-01 00:16:38
999 -1.19  foo  2013-01-01 00:16:39

[1000 rows x 3 columns]

Using an explicit compression type:

In [385]: df.to_pickle("data.pkl.compress", compression="gzip")

In [386]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [387]: rt
Out[387]:
   A     B     C
0 -0.28  foo  2013-01-01 00:00:00
1 -0.08  foo  2013-01-01 00:00:01
2  0.00  foo  2013-01-01 00:00:02
3  1.38  foo  2013-01-01 00:00:03
4  0.34  foo  2013-00:00:04
..   ...   ...   ...
995 -0.22  foo  2013-01-01 00:16:35
996  0.49  foo  2013-01-01 00:16:36
997 -0.46  foo  2013-01-01 00:16:37
998  1.36  foo  2013-01-01 00:16:38
999 -1.19  foo  2013-01-01 00:16:39

[1000 rows x 3 columns]

Inferring compression type from the extension:

In [388]: df.to_pickle("data.pkl.xz", compression="infer")

In [389]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

(continues on next page)
In [390]: rt
Out[390]:

    A    B    C
0 -0.288267  foo 2013-01-01 00:00:00
1 -0.084905  foo 2013-01-01 00:00:01
2  0.004772  foo 2013-01-01 00:00:02
3  1.382989  foo 2013-01-01 00:00:03
4  0.343635  foo 2013-01-01 00:00:04
...     ...     ......
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997  0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

The default is to ‘infer’:

In [391]: df.to_pickle("data.pkl.gz")

In [392]: rt = pd.read_pickle("data.pkl.gz")

In [393]: rt
Out[393]:

    A    B    C
0 -0.288267  foo 2013-01-01 00:00:00
1 -0.084905  foo 2013-01-01 00:00:01
2  0.004772  foo 2013-01-01 00:00:02
3  1.382989  foo 2013-01-01 00:00:03
4  0.343635  foo 2013-01-01 00:00:04
...     ...     ......
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997  0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [394]: df["A"].to_pickle("s1.pkl.bz2")

In [395]: rt = pd.read_pickle("s1.pkl.bz2")

In [396]: rt
Out[396]:

    0       1       2
0 -0.288267
1 -0.084905
2  0.004772
3  1.382989
4  0.343635

(continues on next page)
Passing options to the compression protocol in order to speed up compression:

In [397]: df.to_pickle("data.pkl.gz", compression={"method": "gzip", "compresslevel": 1})

2.4.11 msgpack

pandas support for msgpack has been removed in version 1.0.0. It is recommended to use pickle instead.

Alternatively, you can also the Arrow IPC serialization format for on-the-wire transmission of pandas objects. For documentation on pyarrow, see here.

2.4.12 HDF5 (PyTables)

HDFStore is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the cookbook for some advanced strategies

Warning: pandas uses PyTables for reading and writing HDF5 files, which allows serializing object-dtype data with pickle. Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3/library/pickle.html for more.

In [398]: store = pd.HDFStore("store.h5")

In [399]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

Objects can be written to the file just like adding key-value pairs to a dict:

In [400]: index = pd.date_range("1/1/2000", periods=8)

In [401]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [402]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"])
# store.put('s', s) is an equivalent method
In [403]: store["s"] = s

In [404]: store["df"] = df

In [405]: store
In a current or later Python session, you can retrieve stored objects:

```python
# store.get('df') is an equivalent method
In [406]: store['df']
Out[406]:
```

```
A   B   C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02  -1.613932  1.088104  -0.632963
2000-01-03  -0.585314  -0.275038  -0.937512
2000-01-04   0.632369  -1.249657   0.975593
2000-01-05  -1.060617  -0.143682   0.218423
2000-01-06   3.050329   1.317933  -0.963725
2000-01-07  -0.539452  -0.771133   0.023751
2000-01-08   0.649464  -1.736427   0.197288
```

# dotted (attribute) access provides get as well
In [407]: store.df
Out[407]:
```
A   B   C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02  -1.613932  1.088104  -0.632963
2000-01-03  -0.585314  -0.275038  -0.937512
2000-01-04   0.632369  -1.249657   0.975593
2000-01-05  -1.060617  -0.143682   0.218423
2000-01-06   3.050329   1.317933  -0.963725
2000-01-07  -0.539452  -0.771133   0.023751
2000-01-08   0.649464  -1.736427   0.197288
```

Deletion of the object specified by the key:

# store.remove('df') is an equivalent method
In [408]: del store['df']

In [409]: store
Out[409]:
```
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

In [410]: store.close()

In [411]: store
```
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

In [412]: store.is_open
Out[412]: False

(continues on next page)
# Working with, and automatically closing the store using a context manager

In [413]: with pd.HDFStore("store.h5") as store:
    ....:     store.keys()
    ....:

Read/write API

HDFStore supports a top-level API using read_hdf for reading and to_hdf for writing, similar to how read_csv and to_csv work.

In [414]: df_tl = pd.DataFrame({"A": list(range(5)), "B": list(range(5))})

In [415]: df_tl.to_hdf("store_tl.h5", "table", append=True)

In [416]: pd.read_hdf("store_tl.h5", "table", where="index>2")
Out[416]:
   A  B
  3  3  3
  4  4  4

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting dropna=True.

In [417]: df_with_missing = pd.DataFrame(
    ....:     {
    ....:         "col1": [0, np.nan, 2],
    ....:         "col2": [1, np.nan, np.nan],
    ....:     }
    ....: )
    ....:

In [418]: df_with_missing
Out[418]:
   col1  col2
  0  0.0  1.0
  1  NaN  NaN
  2  2.0  NaN

In [419]: df_with_missing.to_hdf("file.h5", "df_with_missing", format="table", mode="w")

In [420]: pd.read_hdf("file.h5", "df_with_missing")
Out[420]:
   col1  col2
  0  0.0  1.0
  1  NaN  NaN
  2  2.0  NaN

In [421]: df_with_missing.to_hdf(
    ....:     "file.h5", "df_with_missing", format="table", mode="w", dropna=True
    ....: )
    ....: )
In [422]: pd.read_hdf("file.h5", "df_with_missing")
Out[422]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0 NaN</td>
</tr>
</tbody>
</table>

Fixed format

The examples above show storing using `put`, which write the HDF5 to PyTables in a fixed array format, called the fixed format. These types of stores are not appendable once written (though you can simply remove them and rewrite). Nor are they queryable; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The fixed format stores offer very fast writing and slightly faster reading than table stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

```
Warning: A fixed format will raise a TypeError if you try to retrieve using a where:

>>> pd.DataFrame(np.random.randn(10, 2)).to_hdf("test_fixed.h5", "df")
>>> pd.read_hdf("test_fixed.h5", "df", where="index>5")
TypeError: cannot pass a where specification when reading a fixed format. this store must be selected in its entirety
```

Table format

HDFStore supports another PyTables format on disk, the table format. Conceptually a table is shaped very much like a DataFrame, with rows and columns. A table may be appended to in the same or other sessions. In addition, delete and query type operations are supported. This format is specified by `format='table'` or `format='t'` to append or `put` or `to_hdf`.

This format can be set as an option as well `pd.set_option('io.hdf.default_format','table')` to enable `put/append/to_hdf` to by default store in the table format.

In [423]: store = pd.HDFStore("store.h5")

In [424]: df1 = df[0:4]

In [425]: df2 = df[4:]

# append data (creates a table automatically)
In [426]: store.append("df", df1)

In [427]: store.append("df", df2)

In [428]: store
Out[428]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [429]: store.select("df")
Out[429]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000-01-01</td>
<td>1.334065</td>
<td>0.521036</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-02</td>
<td>-1.613932</td>
<td>1.088104</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-03</td>
<td>-0.585314</td>
<td>-0.275038</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-04</td>
<td>0.632369</td>
<td>-1.249657</td>
</tr>
<tr>
<td>5</td>
<td>2000-01-05</td>
<td>1.060617</td>
<td>-0.143682</td>
</tr>
<tr>
<td>6</td>
<td>2000-01-06</td>
<td>3.050329</td>
<td>1.317933</td>
</tr>
<tr>
<td>7</td>
<td>2000-01-07</td>
<td>-0.539452</td>
<td>-0.771133</td>
</tr>
<tr>
<td>8</td>
<td>2000-01-08</td>
<td>0.649464</td>
<td>-1.736427</td>
</tr>
</tbody>
</table>

# the type of stored data
In [430]: store.root.df._v_attrs.pandas_type
Out[430]: 'frame_table'

Note: You can also create a table by passing format='table' or format='t' to a put operation.

Hierarchical keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. foo/bar/bah), which will generate a hierarchy of sub-stores (or Groups in PyTables parlance). Keys can be specified without the leading '/' and are always absolute (e.g. ‘foo’ refers to ‘/foo’). Removal operations can remove everything in the sub-store and below, so be careful.

In [431]: store.put("foo/bar/bah", df)
In [432]: store.append("food/orange", df)
In [433]: store.append("food/apple", df)
In [434]: store
Out[434]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [435]: store.keys()
Out[435]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [436]: store.remove("food")
In [437]: store
Out[437]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5

You can walk through the group hierarchy using the walk method which will yield a tuple for each group key along with the relative keys of its contents.
In [438]: for (path, subgroups, subkeys) in store.walk():
    .....:     for subgroup in subgroups:
    .....:         print("GROUP: {}\n".format(path, subgroup))
    .....:     for subkey in subkeys:
    .....:         key = "/".join([path, subkey])
    .....:         print("KEY: {}\n".format(key))
    .....:         print(store.get(key))

GROUP: /foo
KEY: /df

A  B  C
2000-01-01 1.334065 0.521036 0.930384
2000-01-02 -1.613932 1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04 0.632369 -1.249657 0.975593
2000-01-05 1.060617 -0.143682 0.218423
2000-01-06 3.050329 1.317933 -0.963725
2000-01-07 -0.539452 -0.771133 0.023751
2000-01-08 0.649464 -1.736427 0.197288

GROUP: /foo/bar
KEY: /foo/bar/bah

A  B  C
2000-01-01 1.334065 0.521036 0.930384
2000-01-02 -1.613932 1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04 0.632369 -1.249657 0.975593
2000-01-05 1.060617 -0.143682 0.218423
2000-01-06 3.050329 1.317933 -0.963725
2000-01-07 -0.539452 -0.771133 0.023751
2000-01-08 0.649464 -1.736427 0.197288

Warning: Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

In [8]: store.foo.bar.bah
AttributeError: 'HDFStore' object has no attribute 'foo'

# you can directly access the actual PyTables node but using the root node
In [9]: store.root.foo.bar.bah
Out[9]:
/foo/bar/bah (Group) ''
    children := ['block0_items' (Array), 'block0_values' (Array), 'axis0' (Array),
                 'axis1' (Array)]

Instead, use explicit string based keys:

In [439]: store["foo/bar/bah"]
Out[439]:
A  B  C
2000-01-01 1.334065 0.521036 0.930384
2000-01-02 -1.613932 1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04 0.632369 -1.249657 0.975593
2000-01-05 1.060617 -0.143682 0.218423
2000-01-06 3.050329 1.317933 -0.963725
2000-01-07 -0.539452 -0.771133 0.023751
2000-01-08 0.649464 -1.736427 0.197288
Storing types

Storing mixed types in a table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent attempts at appending longer strings will raise a ValueError.

Passing `min_itemsize={`values`: size}` as a parameter to append will set a larger minimum for the string columns. Storing floats, strings, ints, bools, datetime64 are currently supported. For string columns, passing `nan_rep = 'nan'` to append will change the default nan representation on disk (which converts to/from np. nan), this defaults to nan.

```
in [440]: df_mixed = pd.DataFrame(
       ....:     {  
       ....:         "A": np.random.randn(8),
       ....:         "B": np.random.randn(8),
       ....:         "C": np.array(np.random.randn(8), dtype="float32"),
       ....:         "string": "string",
       ....:         "int": 1,
       ....:         "bool": True,
       ....:         "datetime64": pd.Timestamp("20010102"),
       ....:     },
       ....:     index=list(range(8)),
       ....: )

In [441]: df_mixed.loc[df_mixed.index[3:5], ["A", "B", "string", "datetime64"]] = np.nan
In [442]: store.append("df_mixed", df_mixed, min_itemsize={"values": 50})
In [443]: df_mixed1 = store.select("df_mixed")
In [444]: df_mixed1
Out[444]:
   A         B         C         string          int   bool    datetime64
0 -0.116008  0.743946 -0.398501        string       1   True    2001-01-02
1  0.592375 -0.533097 -0.677311        string       1   True    2001-01-02
2  0.476481 -0.140850 -0.874991        string       1   True    2001-01-02
3    NaN      NaN -1.167564        string       1   True     NaT
4    NaN      NaN -0.593353        string       1   True     NaT
5  0.852727  0.463819  0.146262        string       1   True    2001-01-02
6 -1.177365  0.793644 -0.131959        string       1   True    2001-01-02
7  1.236988  0.221252  0.089012        string       1   True    2001-01-02

In [445]: df_mixed1.dtypes.value_counts()
Out[445]:
float64    2
float32    1
object      1
int64      1
```

(continues on next page)
bool 1
datetime64[ns] 1
dtype: int64

# we have provided a minimum string column size
In [446]: store.root.df_mixed.table
Out[446]: /df_mixed/table (Table(8,))

description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
    "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
    "values_block_2": StringCol(itemsize=50, shape=(1,), dflt=b''),
    "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
    "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
    "values_block_5": Int64Col(shape=(1,), dflt=0, pos=6)
}
byteorder := \'little\'
chunkshape := (689,)
autoindex := True
colindexes := {
    "index": Index(6, mediumshuffle, zlib(1)).is_csi=False

Storing MultiIndex DataFrames

Storing MultiIndex DataFrames as tables is very similar to storing/selecting from homogeneous index DataFrames.

In [447]: index = pd.MultiIndex(
       .....:    levels=["foo", "bar", "baz", "qux"], ["one", "two", "three"]),
       .....:    codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
       .....:    names=["foo", "bar"],
       .....:    )

In [448]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index, columns=["A", "B", "C"])

In [449]: df_mi
Out[449]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>0.667450</td>
<td>0.169405</td>
<td>-1.358046</td>
</tr>
<tr>
<td>two</td>
<td>-0.105563</td>
<td>0.492195</td>
<td>0.076693</td>
</tr>
<tr>
<td>three</td>
<td>0.213685</td>
<td>-0.285283</td>
<td>-1.210529</td>
</tr>
<tr>
<td>bar</td>
<td>-1.408386</td>
<td>0.941577</td>
<td>-0.342447</td>
</tr>
<tr>
<td>two</td>
<td>0.222031</td>
<td>0.052607</td>
<td>2.093214</td>
</tr>
<tr>
<td>baz</td>
<td>1.064908</td>
<td>1.778161</td>
<td>-0.913867</td>
</tr>
<tr>
<td>two</td>
<td>-0.030004</td>
<td>-0.399846</td>
<td>-1.234765</td>
</tr>
<tr>
<td>qux</td>
<td>0.081323</td>
<td>-0.268494</td>
<td>0.168016</td>
</tr>
<tr>
<td>two</td>
<td>-0.898283</td>
<td>-0.218499</td>
<td>1.408028</td>
</tr>
<tr>
<td>three</td>
<td>-1.267828</td>
<td>-0.689263</td>
<td>0.520995</td>
</tr>
</tbody>
</table>

(continues on next page)
In [450]: store.append("df_mi", df_mi)

In [451]: store.select("df_mi")

Out[451]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>one</td>
<td>0.667450</td>
<td>0.169405</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.105563</td>
<td>0.492195</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.213685</td>
<td>-0.285283</td>
</tr>
<tr>
<td></td>
<td>three</td>
<td>-1.150846</td>
<td>-0.342447</td>
</tr>
<tr>
<td>bar</td>
<td></td>
<td>1.408386</td>
<td>0.941577</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.222031</td>
<td>0.052607</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>1.064908</td>
<td>1.778161</td>
</tr>
<tr>
<td></td>
<td>three</td>
<td>-0.030004</td>
<td>-0.399846</td>
</tr>
<tr>
<td>baz</td>
<td></td>
<td>0.222031</td>
<td>0.052607</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.081323</td>
<td>0.268494</td>
</tr>
<tr>
<td></td>
<td>two</td>
<td>0.898283</td>
<td>-0.218499</td>
</tr>
<tr>
<td></td>
<td>three</td>
<td>-1.267828</td>
<td>-0.689263</td>
</tr>
</tbody>
</table>

# the levels are automatically included as data columns

In [452]: store.select("df_mi", "foo=bar")

Out[452]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>one</td>
<td>-1.400836</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two</td>
<td>0.222031</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>1.064908</td>
<td>1.778161</td>
</tr>
<tr>
<td></td>
<td></td>
<td>three</td>
<td>-0.399846</td>
</tr>
<tr>
<td>baz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>one</td>
<td>-0.268494</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two</td>
<td>0.898283</td>
</tr>
<tr>
<td></td>
<td>three</td>
<td>-1.267828</td>
<td>-0.689263</td>
</tr>
</tbody>
</table>

Note: The index keyword is reserved and cannot be use as a level name.

Querying

Querying a table

select and delete operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

A query is specified using the Term class under the hood, as a boolean expression.

- index and columns are supported indexers of DataFrames.
- if data_columns are specified, these can be used as additional indexers.
- level name in a MultiIndex, with default name level_0, level_1,... if not provided.

Valid comparison operators are:

=, ==, !=, >, >=, <, <=

Valid boolean expressions are combined with:

- |: or
- &: and
- ( and ): for grouping
These rules are similar to how boolean expressions are used in pandas for indexing.

**Note:**

- `=` will be automatically expanded to the comparison operator `==`
- `~` is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`
- `"columns in ['A', 'D']"`
- `'columns = A'`
- `'columns == A'`
- `"~(columns = ['A', 'B'])"`
- `'index > df.index[3] & string = "bar"'`
- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The indexers are on the left-hand side of the sub-expression:

```python
columns, major_axis, ts
```

The right-hand side of the sub-expression (after a comparison operator) can be:

- functions that will be evaluated, e.g. `Timestamp('2012-02-01')`
- strings, e.g. "bar"
- date-like, e.g. `20130101`, or "20130101"
- lists, e.g. "['A', 'B']"
- variables that are defined in the local names space, e.g. `date`

**Note:** Passing a string to a query by interpolating it into the query expression is not recommended. Simply assign the string of interest to a variable and use that variable in an expression. For example, do this

```python
string = "HolyMoly"
store.select("df", "index == string")
```

instead of this

```python
string = "HolyMoly"
store.select("df", f"index == {string}")
```

The latter will not work and will raise a `SyntaxError`. Note that there’s a single quote followed by a double quote in the `string` variable.

If you must interpolate, use the `\%r` format specifier
store.select("df", "index == %r" % string)

which will quote string.

Here are some examples:

```
In [453]: dfq = pd.DataFrame(
......:     np.random.randn(10, 4),
......:     columns=list("ABCD"),
......:     index=pd.date_range("20130101", periods=10),
......:     )
......:
In [454]: store.append("dfq", dfq, format="table", data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [455]: store.select("dfq", "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[455]:
   A       B
2013-01-05 -1.083889  0.811865
2013-01-06 -0.402227  1.618922
2013-01-07  0.948196  0.183573
2013-01-08 -1.043530 -0.708145
2013-01-09  0.813949  1.508891
2013-01-10  1.176488 -1.246093
```

Use inline column reference.

```
In [456]: store.select("dfq", where="A>0 or C>0")
Out[456]:
   A       B       C       D
2013-01-01  0.620028  0.159416 -0.263043 -0.639244
2013-01-04 -0.536722  1.005707  0.296917  0.139796
2013-01-05 -1.083889  0.811865  1.648435 -0.164377
2013-01-07  0.948196  0.183573  0.145277  0.308146
2013-01-08 -1.043530 -0.708145  1.430905 -0.850136
2013-01-09  0.813949  1.508891 -1.556154  0.187597
2013-01-10  1.176488 -1.246093 -0.002726 -0.444249
```

The columns keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a 'columns=list_of_columns_to_filter':

```
In [457]: store.select("df", "columns=['A', 'B']")
Out[457]:
   A       B
2000-01-01  1.334065  0.521036
2000-01-02 -1.613932  1.088104
2000-01-03 -0.585314 -0.275038
2000-01-04  0.632369 -1.249657
2000-01-05  1.060617 -0.143682
2000-01-06  3.050329  1.317933
```

(continues on next page)
start and stop parameters can be specified to limit the total search space. These are in terms of the total number of rows in a table.

Note: select will raise a ValueError if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is not a data_column.

select will raise a SyntaxError if the query expression is not valid.

### Query timedelta64[ns]

You can store and query using the timedelta64[ns] type. Terms can be specified in the format: <float>(<unit>), where float may be signed (and fractional), and unit can be D, s, ms, us, ns for the timedelta. Here’s an example:

```python
In [458]: from datetime import timedelta

In [459]: dftd = pd.DataFrame(
       ...:     {
       ...:         "A": pd.Timestamp("20130101"),
       ...:         "B": [pd.Timestamp("20130101") + timedelta(days=i, seconds=10)
       ...:           for i in range(10)]
       ...:     })

In [460]: dftd["C"] = dftd["A"] - dftd["B"]

In [461]: dftd
```

```none
Out[461]:
A       B       C
0 2013-01-01 2013-01-01 00:00:10 -1 days +23:59:50
1 2013-01-01 2013-01-01 00:00:10 -2 days +23:59:50
2 2013-01-01 2013-01-01 00:00:10 -3 days +23:59:50
3 2013-01-01 2013-01-01 00:00:10 -4 days +23:59:50
4 2013-01-01 2013-01-01 00:00:10 -5 days +23:59:50
5 2013-01-01 2013-01-01 00:00:10 -6 days +23:59:50
6 2013-01-01 2013-01-01 00:00:10 -7 days +23:59:50
7 2013-01-01 2013-01-01 00:00:10 -8 days +23:59:50
8 2013-01-01 2013-01-01 00:00:10 -9 days +23:59:50
9 2013-01-01 2013-01-01 00:00:10 -10 days +23:59:50
```

```python
In [462]: store.append("dftd", dftd, data_columns=True)

In [463]: store.select("dftd", "C<-3.5D")
```

```none
Out[463]:
A       B       C
```

(continues on next page)
Query MultiIndex

Selecting from a MultiIndex can be achieved by using the name of the level.

```python
In [464]: df_mi.index.names
Out[464]: FrozenList(['foo', 'bar'])

In [465]: store.select("df_mi", "foo=baz and bar=two")
Out[465]:
   A   B   C
foo baz 1.064908 1.778161 -0.913867
bar two 1.064908 1.778161 -0.913867
```

If the MultiIndex levels names are None, the levels are automatically made available via the level_n keyword with n the level of the MultiIndex you want to select from.

```python
In [466]: index = pd.MultiIndex(levels=["foo", "bar", "baz", "qux"], codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
                           names=["one", "two", "three"],
                          )

In [467]: df_mi_2 = pd.DataFrame(np.random.randn(10, 3), index=index, columns=["A", "B", "C"])

In [468]: df_mi_2
Out[468]:
   A   B   C
foo one 0.856838 1.491776 0.001283
two 0.701816 -1.097917 0.102588
three 0.661740 0.443531 0.559313
bar one -0.459055 -1.22598 -0.455304
two -0.781163 0.826204 -0.530657
baz two 0.296135 1.366810 1.073372
three -0.994957 0.755314 2.119746
qux one -2.628174 -0.089460 -0.133636
two 0.337920 -0.634027 0.421107
three 0.604303 1.053434 1.109990

In [469]: store.append("df_mi_2", df_mi_2)
```

# the levels are automatically included as data columns with keyword level_n
In [470]: store.select("df_mi_2", "level_0=foo and level_1=two")
Out[470]:
   A   B   C
foo baz 1.064908 1.778161 -0.913867
bar two 1.064908 1.778161 -0.913867
```
Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is highly encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`.

**Note:** Indexes are automagically created on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to `append`.

```python
# we have automagically already created an index (in the first section)
In [471]: i = store.root.df.table.cols.index.index

In [472]: i.optlevel, i.kind
Out[472]: (6, 'medium')

# change an index by passing new parameters
In [473]: store.create_table_index("df", optlevel=9, kind="full")

In [474]: i = store.root.df.table.cols.index.index

In [475]: i.optlevel, i.kind
Out[475]: (9, 'full')
```

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```python
In [476]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [477]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [478]: st = pd.HDFStore("appends.h5", mode="w")

In [479]: st.append("df", df_1, data_columns=["B"], index=False)

In [480]: st.append("df", df_2, data_columns=["B"], index=False)

In [481]: st.get_storer("df").table
Out[481]:
```/df/table (Table(20,)) ''
```
description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
byteorder := 'little'
chunkshape := (2730,)
```

Then create the index when finished appending.
In [482]: st.create_table_index("df", columns="B", optlevel=9, kind="full")

In [483]: st.get_storer("df").table
Out[483]:
/df/table (Table(20,)) '
   description := {
      "index": Int64Col(shape=(), dflt=0, pos=0),
      "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
      "B": Float64Col(shape=(), dflt=0.0, pos=2)}
   byteorder := 'little'
   chunkshape := (2730,)
   autoindex := True
   colindexes := {
      "B": Index(9, fullshuffle, zlib(1)).is CSI=True}

In [484]: st.close()

See here for how to create a completely-sorted-index (CSI) on an existing store.

Query via data columns

You can designate (and index) certain columns that you want to be able to perform queries (other than the indexable columns, which you can always query). For instance say you want to perform this common operation, on-disk, and return just the frame that matches this query. You can specify data_columns = True to force all columns to be data_columns.

In [485]: df_dc = df.copy()

In [486]: df_dc["string"] = "foo"

In [487]: df_dc.loc[df_dc.index[4:6], "string"] = np.nan

In [488]: df_dc.loc[df_dc.index[7:9], "string"] = "bar"

In [489]: df_dc["string2"] = "cool"

In [490]: df_dc.loc[df_dc.index[1:3], ["B", "C"] ] = 1.0

In [491]: df_dc
Out[491]:
   A       B       C string string2
2000-01-01 1.334065 0.521036 0.930384 foo   cool
2000-01-02 -1.613932 1.000000 1.000000 foo   cool
2000-01-03 -0.585314 1.000000 1.000000 foo   cool
2000-01-04 0.632369 -1.249657 0.975593 foo   cool
2000-01-05 1.060617 -0.143682 0.218423 NaN   cool
2000-01-06 3.050329 1.317933 -0.963725 NaN   cool
2000-01-07 -0.539452 -0.771133 0.023751 foo   cool
2000-01-08 0.649464 -1.736427 0.197288 bar   cool

# on-disk operations
In [492]: store.append("df_dc", df_dc, data_columns=["B", "C", "string", "string2")

(continues on next page)
In [493]: store.select("df_dc", where="B > 0")
Out[493]:
          A       B       C  string  string2
2000-01-01  1.334065  0.521036  0.930384  foo     cool
2000-01-02 -1.613932  1.000000  1.000000  foo     cool
2000-01-03 -0.585314  1.000000  1.000000  foo     cool
2000-01-06  3.050329  1.317933 -0.963725   NaN    cool

# getting creative
In [494]: store.select("df_dc", "B > 0 & C > 0 & string == foo")
Out[494]:
          A       B       C  string  string2
2000-01-01  1.334065  0.521036  0.930384  foo     cool
2000-01-02 -1.613932  1.000000  1.000000  foo     cool
2000-01-03 -0.585314  1.000000  1.000000  foo     cool

# this is in-memory version of this type of selection
In [495]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == "foo")]
Out[495]:
          A       B       C  string  string2
2000-01-01  1.334065  0.521036  0.930384  foo     cool
2000-01-02 -1.613932  1.000000  1.000000  foo     cool
2000-01-03 -0.585314  1.000000  1.000000  foo     cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as `PyTables` columns
In [496]: store.root.df_dc.table
Out[496]:
/df_dc/table (Table(8,)) ''
description := {
"index": Int64Col(shape=(), dflt=0, pos=0),
"values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
"B": Float64Col(shape=(), dflt=0.0, pos=2),
"C": Float64Col(shape=(), dflt=0.0, pos=3),
"string": StringCol(itemsize=3, shape=(), dflt=b''),
"string2": StringCol(itemsize=4, shape=(), dflt=b''},
byteorder := 'little'
chunkshape := (1680,)
autoindex := True
colindexes := {
"index": Index(6, mediumshuffle, zlib(1)).is_csi=False,
"B": Index(6, mediumshuffle, zlib(1)).is_csi=False,
"C": Index(6, mediumshuffle, zlib(1)).is_csi=False,
"string": Index(6, mediumshuffle, zlib(1)).is_csi=False,
"string2": Index(6, mediumshuffle, zlib(1)).is_csi=False}

There is some performance degradation by making lots of columns into data columns, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!).
Iterator

You can pass `iterator=True` or `chunksize=number_in_a_chunk` to `select` and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```
In [497]: for df in store.select("df", chunksize=3):
    .....:   print(df)
    .....:
           A    B    C
2000-01-01  1.334065 0.521036 0.930384
2000-01-02 -1.613932 1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
           A    B    C
2000-01-04  0.632369 -1.249657 0.975593
2000-01-05  1.060617 -0.143682 0.218423
2000-01-06  3.050329  1.317933 -0.963725
           A    B    C
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
```

**Note:** You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in pd.read_hdf("store.h5", "df", chunksize=3):
    print(df)
```

Note, that the chunksize keyword applies to the `source` rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```
In [498]: dfeq = pd.DataFrame({"number": np.arange(1, 11)})

In [499]: dfeq
Out[499]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
9      10

In [500]: store.append("dfeq", dfeq, data_columns=['number'])

In [501]: def chunks(l, n):
    .....:   return [l[i: i + n] for i in range(0, len(l), n)]
```
In [502]: evens = [2, 4, 6, 8, 10]

In [503]: coordinates = store.select_as_coordinates("dfeq", "number=evens")

In [504]: for c in chunks(coordinates, 2):
....:
    print(store.select("dfeq", where=c))
    number
     1  2
     3  4
     5  6
     7  8
     9 10

Advanced queries

Select a single column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

In [505]: store.select_column("df_dc", "index")

Out[505]:
0  2000-01-01
1  2000-01-02
2  2000-01-03
3  2000-01-04
4  2000-01-05
5  2000-01-06
6  2000-01-07
7  2000-01-08
Name: index, dtype: datetime64[ns]

In [506]: store.select_column("df_dc", "string")

Out[506]:
0  foo
1  foo
2  foo
3  foo
4  NaN
5  NaN
6  foo
7  bar
Name: string, dtype: object
Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [507]: df_coord = pd.DataFrame(
       ....:     np.random.randn(1000, 2), index=pd.date_range("20000101", periods=1000)
       ....: )
       ....:

In [508]: store.append("df_coord", df_coord)

In [509]: c = store.select_as_coordinates("df_coord", "index > 20020101")

In [510]: c
Out[510]: Int64Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
    ... 990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
    dtype='int64', length=268)

In [511]: store.select("df_coord", where=c)
Out[511]:
   0         1
2002-01-02 -0.165548  0.646989
2002-01-03  0.782753 -0.123409
2002-01-04 -0.391932 -0.740915
2002-01-05  1.211070 -0.668715
2002-01-06  0.341987 -0.685867
    ...     ...     ...
2002-09-22  1.788110 -0.405908
2002-09-23 -0.801912  0.768460
2002-09-24  0.466284 -0.457411
2002-09-25 -0.364060  0.785367
2002-09-26 -1.463093  1.187315
[268 rows x 2 columns]
```

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this mask would be a resulting index from an indexing operation. This example selects the months of a datetimeindex which are 5.

```
In [512]: df_mask = pd.DataFrame(
       ....:     np.random.randn(1000, 2), index=pd.date_range("20000101", periods=1000)
       ....: )
       ....:

In [513]: store.append("df_mask", df_mask)

In [514]: c = store.select_column("df_mask", "index")

(continues on next page)
In [515]: where = c[pd.DatetimeIndex(c).month == 5].index

In [516]: store.select("df_mask", where=where)

Out[516]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-05-01</td>
<td>1.735883 -2.615261</td>
</tr>
<tr>
<td>2000-05-02</td>
<td>0.422173  2.425154</td>
</tr>
<tr>
<td>2000-05-03</td>
<td>0.632453 -0.165640</td>
</tr>
<tr>
<td>2000-05-04</td>
<td>-1.017207 -0.005696</td>
</tr>
<tr>
<td>2000-05-05</td>
<td>0.299606  0.070606</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2002-05-27</td>
<td>0.234503  1.199126</td>
</tr>
<tr>
<td>2002-05-28</td>
<td>-3.021833 -1.016828</td>
</tr>
<tr>
<td>2002-05-29</td>
<td>0.522794  0.063465</td>
</tr>
<tr>
<td>2002-05-30</td>
<td>-1.653736 0.031709</td>
</tr>
<tr>
<td>2002-05-31</td>
<td>-0.968402 -0.393583</td>
</tr>
</tbody>
</table>

[93 rows x 2 columns]

Storer object

If you want to inspect the stored object, retrieve via get_storer. You could use this programmatically to say get the number of rows in an object.

In [517]: store.get_storer("df_dc").nrows

Out[517]: 8

Multiple table queries

The methods append_to_multiple and select_as_multiple can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table’s index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

The append_to_multiple method splits a given single DataFrame into multiple tables according to d, a dictionary that maps the table names to a list of ‘columns’ you want in that table. If None is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument selector defines which table is the selector table (which you can make queries from). The argument dropna will drop rows from the input DataFrame to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely np.Nan, that row will be dropped from all tables.

If dropna is False, THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES. Remember that entirely np.Nan rows are not written to the HDFStore, so if you choose to call dropna=False, some tables may have more rows than others, and therefore select_as_multiple may not work or it may return unexpected results.

In [518]: df_mt = pd.DataFrame(
      ...: np.random.randn(8, 6),
      ...: index=pd.date_range("1/1/2000", periods=8),
      ...: columns=["A", "B", "C", "D", "E", "F"],
      ...
      ...: )

(continues on next page)
In [519]: df_mt['foo'] = 'bar'

In [520]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [521]: store.append_to_multiple(
    ....:   {"df1_mt": ["A", "B"], "df2_mt": None}, df_mt, selector="df1_mt"
    ....:   )
    ....:

In [522]: store
Out[522]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [523]: store.select("df1_mt")
Out[523]:
      A  B
2000-01-01  1.251079 -0.362628
2000-01-02   NaN     NaN
2000-01-03  0.719421 -0.448886
2000-01-04  1.140998 -0.877922
2000-01-05  1.043605  1.798494
2000-01-06 -0.467812 -0.027965
2000-01-07  0.150568  0.754820
2000-01-08 -0.596306 -0.910022

In [524]: store.select("df2_mt")
Out[524]:
     A  B  C  D  E  F  foo
2000-01-01  1.602451 -0.221229  0.712403  0.465927 bar
2000-01-02 -0.525571  0.851566 -0.681308 -0.549386 bar
2000-01-03 -0.044171  1.396628  1.041242 -1.588171 bar
2000-01-04  0.463351 -0.861042 -2.192841 -1.025263 bar
2000-01-05 -1.954845 -1.712882 -2.757922 -0.307713 bar
2000-01-06  1.601542 -0.417884 -0.307713  1.732966 bar
2000-01-07 -1.935461  1.007668  0.079529 -1.459471 bar
2000-01-08 -1.057072 -0.864360 -1.124870  1.732966 bar

# as a multiple
In [525]: store.select_as_multiple(
    ....:   ["df1_mt", "df2_mt"],
    ....:   where=["A>0", "B>0"],
    ....:   selector="df1_mt",
    ....:   )
    ....:
Out[525]:
      A  B  C  D  E  F  foo
Delete from a table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the PyTables deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very expensive operation depending on the orientation of your data. To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the `indexables`.

Data is ordered (on the disk) in terms of the `indexables`. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- `date_1`
  - `id_1`
  - `id_2`
  - ...
  - `id_n`

- `date_2`
  - `id_1`
  - ...
  - `id_n`

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is removed, then the following data moved. On the other hand a delete operation on the `minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

**Warning:** Please note that HDF5 **DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again, **WILL TEND TO INCREASE THE FILE SIZE**.

To repack and clean the file, use `ptrepack`.

Notes & caveats

Compression

PyTables allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

- `complevel` specifies if and how hard data is to be compressed. `complevel=0` and `complevel=None` disables compression and `0<complevel<10` enables compression.

- `complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:
  - `zlib`: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
- lzo: Fast compression and decompression.
- bzip2: Good compression rates.
- blosc: Fast compression and decompression.

Support for alternative blosc compressors:

* blosc:blosclz This is the default compressor for blosc
* blosc:lz4: A compact, very popular and fast compressor.
* blosc:lz4hc: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
* blosc:snappy: A popular compressor used in many places.
* blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
* blosc:zstd: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

**Note:** If the library specified with the `complib` option is missing on your platform, compression defaults to zlib without further ado.

Enable compression for all objects within the file:

```python
store_compressed = pd.HDFStore(
    "store_compressed.h5", complevel=9, complib="blosc:blosclz"
)
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```python
store.append("df", df, complib="zlib", complevel=5)
```

**ptrepack**

PyTables offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied PyTables utility ptrepack. In addition, ptrepack can change compression levels after the fact.

```bash
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore ptrepack in.h5 out.h5 will repack the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the copy method.
Caveats

**Warning:** **HDFStore is not-threadsafe for writing.** The underlying PyTables only supports concurrent reads (via threading or processes). If you need reading and writing **at the same time**, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the [GH2397](https://github.com/pandas-dev/pandas/issues/2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.

- Once a table is created columns (DataFrame) are fixed; only exactly the same columns can be appended

- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

**Warning:** **PyTables will show a NaturalNameWarning** if a column name cannot be used as an attribute selector. **Natural** identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a **where** clause and are generally a bad idea.

**DataTypes**

**HDFStore** will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

<table>
<thead>
<tr>
<th>Type</th>
<th>Represents missing values</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating: float64, float32, float16</td>
<td>np.nan</td>
</tr>
<tr>
<td>integer: int64, int32, int8, uint64, uint32, uint8</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>NaT</td>
</tr>
<tr>
<td>datetime64[ns]</td>
<td>NaT</td>
</tr>
<tr>
<td>timedelta64[ns]</td>
<td>NaT</td>
</tr>
<tr>
<td>categorical : see the section below</td>
<td></td>
</tr>
<tr>
<td>object: strings</td>
<td>np.nan</td>
</tr>
</tbody>
</table>

Unicode columns are not supported, and **WILL FAIL**.

**Categorical data**

You can write data that contains category dtypes to a **HDFStore**. Queries work the same as if it was an object array. However, the category dtyped data is stored in a more efficient manner.

```python
In [526]: dfcat = pd.DataFrame(
    ....:     {"A": pd.Series(list("aabbcdba")).astype("category"), "B": np.random.randn(8)}
    ....: )
    ....: 
```

(continues on next page)
In [527]: dfcat
Out[527]:
   A   B
0  a  0.477849
1  a  0.283128
2  b -2.045700
3  b  0.338206
4  c -0.423113
5  d  2.314361
6  b -0.033100
7  a -0.965461

In [528]: dfcat.dtypes
Out[528]:
A    category
B    float64
dtype: object

In [529]: cstore = pd.HDFStore("cats.h5", mode="w")
In [530]: cstore.append("dfcat", dfcat, format="table", data_columns=["A"])
In [531]: result = cstore.select("dfcat", where="A in ['b', 'c']")
In [532]: result
Out[532]:
   A   B
2  b -2.045700
3  b  0.338206
4  c -0.423113
6  b -0.033100

In [533]: result.dtypes
Out[533]:
A    category
B    float64
dtype: object

String columns

min_itemsize

The underlying implementation of HDFStore uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the HDFStore, in the first append. Subsequent appends, may introduce a string for a column larger than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass min_itemsize on the first table creation to a-priori specify the minimum length of a particular string column. min_itemsize can be an integer, or a dict mapping a column name to an integer. You can pass values as a key to allow all indexables or data_columns to have this min_itemsize.

Passing a min_itemsize dict will cause all passed columns to be created as data_columns automatically.
**Note:** If you are not passing any `data_columns`, then the `min_itemsize` will be the maximum of the length of any string passed

```python
In [534]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'}, index=list(range(5)))

In [535]: dfs
Out[535]:
   A  B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
4  foo  bar

# A and B have a size of 30
In [536]: store.append("dfs", dfs, min_itemsize=30)

In [537]: store.get_storer("dfs").table
Out[537]:
/dfs/table (Table(5,)) ''
   description := {
"index": Int64Col(shape=(), dflt=0, pos=0),
"values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
   byteorder := 'little'
   chunkshape := (963,)
   autoindex := True
   colindexes := {
   "index": Index(6, mediumshuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [538]: store.append("dfs2", dfs, min_itemsize={'A': 30})

In [539]: store.get_storer("dfs2").table
Out[539]:
/dfs2/table (Table(5,)) ''
   description := {
"index": Int64Col(shape=(), dflt=0, pos=0),
"values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
"A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
   byteorder := 'little'
   chunkshape := (1598,)
   autoindex := True
   colindexes := {
   "index": Index(6, mediumshuffle, zlib(1)).is_csi=False,
   "A": Index(6, mediumshuffle, zlib(1)).is_csi=False}
```

**nan_rep**

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.
In [540]: dfss = pd.DataFrame({'A': ['foo', 'bar', 'nan']})

In [541]: dfss
Out[541]:
    A
0  foo
1  bar
2  nan

In [542]: store.append("dfss", dfss)

In [543]: store.select("dfss")
Out[543]:
    A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [544]: store.append("dfss2", dfss, nan_rep="_nan_")

In [545]: store.select("dfss2")
Out[545]:
    A
0  foo
1  bar
2  nan

External compatibility

HDFStore writes table format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, HDFStore can read native PyTables format tables.

It is possible to write an HDFStore object that can easily be imported into R using the rhdf5 library (Package website). Create a table format store like this:

In [546]: df_for_r = pd.DataFrame({
                     "first": np.random.rand(100),
                     "second": np.random.rand(100),
                     "class": np.random.randint(0, 2, (100,))},
                     index=range(100),
                     )

In [547]: df_for_r.head()
Out[547]:
         first second  class
0  0.864919  0.852910     0
1  0.039579  0.412962     1
2  0.015226  0.978410     0

(continues on next page)
In [548]: store_export = pd.HDFStore("export.h5")

In [549]: store_export.append("df_for_r", df_for_r, data_columns=df_dc.columns)

In [550]: store_export
Out[550]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5

In R this file can be read into a data.frame object using the rhdf5 library. The following example function reads the corresponding column names and data values from the values and assembles them into a data.frame:

```r
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.
library(rhdf5)
loadhdf5data <- function(h5File) {

  listing <- h5ls(h5File)
  # Find all data nodes, values are stored in *_values and corresponding column
  # titles in *_items
  data_nodes <- grep("_values", listing$name)
  name_nodes <- grep("_items", listing$name)
  data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
  name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
  columns = list()
  for (idx in seq(data_paths)) {
    # NOTE: matrices returned by h5read have to be transposed to obtain
    # required Fortran order!
    data <- data.frame(t(h5read(h5File, data_paths[idx])))
    names <- t(h5read(h5File, name_paths[idx]))
    entry <- data.frame(data)
    colnames(entry) <- names
    columns <- append(columns, entry)
  }

  data <- data.frame(columns)

  return(data)
}
```

Now you can import the DataFrame into R:

```r
> data = loadhdf5data("transfer.hdf5")
> head(data)
   first    second    class
1 0.417022 0.326645  0
2 0.720324 0.527058  0

(continues on next page)
```
Performance

- **tables** format come with a writing performance penalty as compared to fixed stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedRows=<int>` to the first `append`, to set the TOTAL number of rows that PyTables will expect. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A PerformanceWarning will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See [Here](#) for more information and some solutions.

### 2.4.13 Feather

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize DataFrames, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats:

- The format will NOT write an Index, or MultiIndex for the DataFrame and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported
- Actual Python objects in object dtype columns are not supported. These will raise a helpful error message on an attempt at serialization.

See the [Full Documentation](#).

```
In [551]: df = pd.DataFrame(
    ....:     {  
    ....:         "a": list("abc"),  
    ....:         "b": list(range(1, 4)),  
    ....:         "c": np.arange(3, 6).astype("u1"),  
    ....:         "d": np.arange(4.0, 7.0, dtype="float64"),  
    ....:         "e": [True, False, True],
```

(continues on next page)
In 

```python
In [552]: df
Out[552]:
   a  b  c  d  e  f  g  h
0  a  1  3  4.0  True  a  2013-01-01  2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
   ...00000000  
1  b  2  4  5.0  False  b  2013-01-02  2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
   ...000000001  
2  c  3  5  6.0  True  c  2013-01-03  2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
   ...000000002
```

In 

```python
In [553]: df.dtypes
Out[553]:
a object
b int64
c uint8
d float64
e bool
f category
g datetime64[ns]
h datetime64[ns, US/Eastern]
i datetime64[ns]
dtype: object
```

Write to a feather file.

```python
In [554]: df.to_feather("example.feather")
```

Read from a feather file.

```python
In [555]: result = pd.read_feather("example.feather")

In [556]: result
Out[556]:
   a  b  c  d  e  f  g  h
0  a  1  3  4.0  True  a  2013-01-01  2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
   ...00000000  
1  b  2  4  5.0  False  b  2013-01-02  2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
   ...000000001  
2  c  3  5  6.0  True  c  2013-01-03  2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
   ...000000002
```

# we preserve dtypes
In [557]: result.dtypes
Out[557]:
a object
b int64
c uint8
d float64
e bool
f category
g datetime64[ns]
h datetime64[ns, US/Eastern]
i datetime64[ns]
dtype: object

2.4.14 Parquet

Apache Parquet provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize DataFrame s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- The pyarrow engine always writes the index to the output, but fastparquet only writes non-default indexes. This extra column can cause problems for non-pandas consumers that are not expecting it. You can force including or omitting indexes with the index argument, regardless of the underlying engine.
- Index level names, if specified, must be strings.
- In the pyarrow engine, categorical dtypes for non-string types can be serialized to parquet, but will de-serialize as their primitive dtype.
- The pyarrow engine preserves the ordered flag of categorical dtypes with string types. fastparquet does not preserve the ordered flag.
- Non supported types include Interval and actual Python object types. These will raise a helpful error message on an attempt at serialization. Period type is supported with pyarrow >= 0.16.0.
- The pyarrow engine preserves extension data types such as the nullable integer and string data type (requiring pyarrow >= 0.16.0, and requiring the extension type to implement the needed protocols, see the extension types documentation).

You can specify an engine to direct the serialization. This can be one of pyarrow, or fastparquet, or auto. If the engine is NOT specified, then the pd.options.io.parquet.engine option is checked; if this is also auto, then pyarrow is tried, and falling back to fastparquet.

See the documentation for pyarrow and fastparquet.

Note: These engines are very similar and should read/write nearly identical parquet format files. Currently pyarrow does not support timedelta data, fastparquet>=0.1.4 supports timezone aware datetimes. These libraries differ by having different underlying dependencies (fastparquet by using numba, while pyarrow uses a c-library).
In [558]: df = pd.DataFrame({
          ....:     "a": list("abc"),
          ....:     "b": list(range(1, 4)),
          ....:     "c": np.arange(3, 6).astype("u1"),
          ....:     "d": np.arange(4.0, 7.0, dtype="float64"),
          ....:     "e": [True, False, True],
          ....:     "f": pd.date_range("20130101", periods=3),
          ....:     "g": pd.date_range("20130101", periods=3, tz="US/Eastern"),
          ....:     "h": pd.Categorical(list("abc")),
          ....:     "i": pd.Categorical(list("abc"), ordered=True),
          ....:  })
In [559]: df
Out[559]:
   a  b  c  d  e     f     g     h     i
0  a  1  3  4.0  True  2013-01-01 00:00:00-05:00 a  a
1  b  2  4  5.0 False  2013-01-02 00:00:00-05:00 b  b
2  c  3  5  6.0  True  2013-01-03 00:00:00-05:00 c  c

In [560]: df.dtypes
Out[560]:
a     object
b    int64
c   uint8
d  float64
e      bool
f   datetime64[ns]
g  datetime64[ns, US/Eastern]
h      category
i      category
dtype: object

Write to a parquet file.

In [561]: df.to_parquet("example_pa.parquet", engine="pyarrow")
In [562]: df.to_parquet("example_fp.parquet", engine="fastparquet")

Read from a parquet file.

In [563]: result = pd.read_parquet("example_fp.parquet", engine="fastparquet")
In [564]: result = pd.read_parquet("example_pa.parquet", engine="pyarrow")
In [565]: result.dtypes
Out[565]:
a     object
b    int64
c   uint8
d  float64
(continues on next page)
Read only certain columns of a parquet file.

```python
In [566]: result = pd.read_parquet(  
......:   "example_fp.parquet",  
......:   engine="fastparquet",  
......:   columns=["a", "b"],  
......: )
......:
In [567]: result = pd.read_parquet(  
......:   "example_pa.parquet",  
......:   engine="pyarrow",  
......:   columns=["a", "b"],  
......: )
......:
In [568]: result.dtypes
Out[568]:
a  object
b  int64
dtype: object
```

**Handling indexes**

Serializing a DataFrame to parquet may include the implicit index as one or more columns in the output file. Thus, this code:

```python
In [569]: df = pd.DataFrame({"a": [1, 2], "b": [3, 4]})
In [570]: df.to_parquet("test.parquet", engine="pyarrow")
```

creates a parquet file with *three* columns if you use *pyarrow* for serialization: a, b, and __index_level_0__. If you're using *fastparquet*, the index may or may not be written to the file.

This unexpected extra column causes some databases like Amazon Redshift to reject the file, because that column doesn't exist in the target table.

If you want to omit a dataframe's indexes when writing, pass `index=False` to `to_parquet()`:

```python
In [571]: df.to_parquet("test.parquet", index=False)
```

This creates a parquet file with just the two expected columns, a and b. If your DataFrame has a custom index, you won't get it back when you load this file into a DataFrame.

Passing `index=True` will *always* write the index, even if that's not the underlying engine's default behavior.
Partitioning Parquet files

Parquet supports partitioning of data based on the values of one or more columns.

```python
In [572]: df = pd.DataFrame({"a": [0, 0, 1, 1], "b": [0, 1, 0, 1]})
In [573]: df.to_parquet(path="test", engine="pyarrow", partition_cols=['a'],
    compression=None)
```

The path specifies the parent directory to which data will be saved. The `partition_cols` are the column names by which the dataset will be partitioned. Columns are partitioned in the order they are given. The partition splits are determined by the unique values in the partition columns. The above example creates a partitioned dataset that may look like:

```
2.4.15 ORC

New in version 1.0.0.

Similar to the parquet format, the ORC Format is a binary columnar serialization for data frames. It is designed to make reading data frames efficient. pandas provides only a reader for the ORC format, `read_orc()`. This requires the pyarrow library.

Warning:

- It is highly recommended to install pyarrow using conda due to some issues occurred by pyarrow.
- `read_orc()` is not supported on Windows yet, you can find valid environments on install optional dependencies.

2.4.16 SQL queries

The pandas.io.sql module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python’s standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some cookbook examples for some advanced strategies.

The key functions are:
read_sql_table(table_name, con[, schema, ...])  Read SQL database table into a DataFrame.
read_sql_query(sql, con[, index_col, ...])  Read SQL query into a DataFrame.
read_sql(sql, con[, index_col, ...])  Read SQL query or database table into a DataFrame.
DataFrame.to_sql(name, con[, schema, ...])  Write records stored in a DataFrame to a SQL database.

Note:  The function read_sql() is a convenience wrapper around read_sql_table() and read_sql_query() (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the SQLite SQL database engine. You can use a temporary SQLite database where data are stored in “memory”.

To connect with SQLAlchemy you use the create_engine() function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on create_engine() and the URI formatting, see the examples below and the SQLAlchemy documentation:

```python
In [574]: from sqlalchemy import create_engine
    # Create your engine.
In [575]: engine = create_engine("sqlite:///:memory:")
```

If you want to manage your own connections you can pass one of those instead. The example below opens a connection to the database using a Python context manager that automatically closes the connection after the block has completed. See the SQLAlchemy docs for an explanation of how the database connection is handled:

```python
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table("data", conn)
```

Warning:  When you open a connection to a database you are also responsible for closing it. Side effects of leaving a connection open may include locking the database or other breaking behaviour.

Writing DataFrames

Assuming the following data is in a DataFrame `data`, we can insert it into the database using `to_sql()`:

```
<table>
<thead>
<tr>
<th>id</th>
<th>Date</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>2012-10-18</td>
<td>X</td>
<td>25.7</td>
<td>True</td>
</tr>
<tr>
<td>42</td>
<td>2012-10-19</td>
<td>Y</td>
<td>-12.4</td>
<td>False</td>
</tr>
<tr>
<td>63</td>
<td>2012-10-20</td>
<td>Z</td>
<td>5.73</td>
<td>True</td>
</tr>
</tbody>
</table>
```

```python
In [576]: data
Out[576]:
    id    Date  Col_1  Col_2  Col_3
0  26  2012-10-18 X    25.70  True
1  42  2012-10-19 Y   -12.40 False
2  63  2012-10-20 Z    5.730  True
```

```python
In [577]: data.to_sql("data", engine)
Out[577]: 3
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the chunksize parameter when calling `to_sql`. For example, the following writes data to the database in batches of 1000 rows at a time:

```python
In [578]: data.to_sql("data_chunked", engine, chunksize=1000)
Out[578]: 3
```

### SQL data types

to_sql() will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype `object`, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the dtype argument. This argument needs a dictionary mapping column names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```python
In [579]: from sqlalchemy.types import String
In [580]: data.to_sql("data_dtype", engine, dtype={"Col_1": String})
Out[580]: 3
```

**Note:** Due to the limited support for timedelta’s in the different database flavors, columns with type `timedelta64` will be written as integer values as nanoseconds to the database and a warning will be raised.

**Note:** Columns of `category` dtype will be converted to the dense representation as you would get with `np.asarray(categorical)` (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does not generate a categorical.

### Datetime data types

Using SQLAlchemy, to_sql() is capable of writing datetime data that is timezone naive or timezone aware. However, the resulting data stored in the database ultimately depends on the supported data type for datetime data of the database system being used.

The following table lists supported data types for datetime data for some common databases. Other database dialects may have different data types for datetime data.

<table>
<thead>
<tr>
<th>Database</th>
<th>SQL Datetime Types</th>
<th>Timezone Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>TEXT</td>
<td>No</td>
</tr>
<tr>
<td>MySQL</td>
<td>TIMESTAMP or DATETIME</td>
<td>No</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>TIMESTAMP or TIMESTAMP WITH TIME ZONE</td>
<td>Yes</td>
</tr>
</tbody>
</table>

When writing timezone aware data to databases that do not support timezones, the data will be written as timezone naive timestamps that are in local time with respect to the timezone.

read_sql_table() is also capable of reading datetime data that is timezone aware or naive. When reading `TIMESTAMP WITH TIME ZONE` types, pandas will convert the data to UTC.
**Insertion method**

The parameter `method` controls the SQL insertion clause used. Possible values are:

- **None**: Uses standard SQL INSERT clause (one per row).

- **'multi'**: Pass multiple values in a single INSERT clause. It uses a special SQL syntax not supported by all backends. This usually provides better performance for analytic databases like Presto and Redshift, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documentation.

- **callable with signature (pd_table, conn, keys, data_iter)**: This can be used to implement a more performant insertion method based on specific backend dialect features.

Example of a callable using PostgreSQL COPY clause:

```python
# Alternative to _sql() *method* for DBs that support COPY FROM
import csv
from io import StringIO

def psql_insert_copy(table, conn, keys, data_iter):
    """
    Execute SQL statement inserting data
    Parameters
    ----------
    table : pandas.io.sql.SQLTable
    conn : sqlalchemy.engine.Engine or sqlalchemy.engine.Connection
    keys : list of str
        Column names
    data_iter : Iterable that iterates the values to be inserted
    """
    # gets a DBAPI connection that can provide a cursor
dbapi_conn = conn.connection
    with dbapi_conn.cursor() as cur:
        s_buf = StringIO()
        writer = csv.writer(s_buf)
        writer.writerows(data_iter)
        s_buf.seek(0)
        columns = ', '.join(["{}").format(k) for k in keys])
        if table.schema:
            table_name = '{}.{}.format(table.schema, table.name)
        else:
            table_name = table.name
        sql = 'COPY {} {} FROM STDIN WITH CSV'.format(
            table_name, columns)
        cur.copy_expert(sql=sql, file=s_buf)
```
### Reading tables

`read_sql_table()` will read a database table given the table name and optionally a subset of columns to read.

**Note:** In order to use `read_sql_table()`, you **must** have the SQLAlchemy optional dependency installed.

```python
In [581]: pd.read_sql_table("data", engine)
Out[581]:
     index  id     Date  Col_1  Col_2  Col_3
0      0   26  2010-10-18     X    27.50   True
1      1   42  2010-10-19     Y   -12.50   False
2      2   63  2010-10-20     Z     5.73   True
```

**Note:** Note that pandas infers column dtypes from query outputs, and not by looking up data types in the physical database schema. For example, assume `userid` is an integer column in a table. Then, intuitively, `select userid` .. . will return integer-valued series, while `select cast(userid as text) ...` will return object-valued (str) series. Accordingly, if the query output is empty, then all resulting columns will be returned as object-valued (since they are most general). If you foresee that your query will sometimes generate an empty result, you may want to explicitly typecast afterwards to ensure dtype integrity.

You can also specify the name of the column as the DataFrame index, and specify a subset of columns to be read.

```python
In [582]: pd.read_sql_table("data", engine, index_col= "id")
Out[582]:
     index     Date  Col_1  Col_2  Col_3
   id
 26   0  2010-10-18     X    27.50   True
 42   1  2010-10-19     Y   -12.50   False
 63   2  2010-10-20     Z     5.73   True
```

```python
In [583]: pd.read_sql_table("data", engine, columns= ["Col_1", "Col_2"])
Out[583]:
     Col_1  Col_2
0       X    27.50
1       Y   -12.50
2       Z     5.73
```

And you can explicitly force columns to be parsed as dates:

```python
In [584]: pd.read_sql_table("data", engine, parse_dates=[ "Date"])
Out[584]:
     index  id     Date  Col_1  Col_2  Col_3
0      0   26  2010-10-18     X    27.50   True
1      1   42  2010-10-19     Y   -12.50   False
2      2   63  2010-10-20     Z     5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to `pandas.to_datetime()`:

```python
df = pd.read_sql_table("data", engine, parse_dates= ["Date"])
df = pd.read_sql_table("data", engine, parse_dates= ["%Y-%m-%d"])
```

(continues on next page)
You can check if a table exists using `has_table()`

**Schema support**

Reading from and writing to different schema’s is supported through the schema keyword in the `read_sql_table()` and `to_sql()` functions. Note however that this depends on the database flavor (sqlite does not have schema’s). For example:

```python
df.to_sql("table", engine, schema="other_schema")
pd.read_sql_table("table", engine, schema="other_schema")
```

**Querying**

You can query using raw SQL in the `read_sql_query()` function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```python
In [585]: pd.read_sql_query("SELECT * FROM data", engine)
Out[585]:
     index  id         Date  Col_1  Col_2  Col_3
0      0   26 2010-10-18        X  27.50    1
1      1  42 2010-10-19       Y -12.50    0
2      2  63 2010-10-20       Z   5.73    1
```

Of course, you can specify a more “complex” query.

```python
In [586]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engine)
Out[586]:
   id  Col_1  Col_2
0  42      Y   -12.5
```

The `read_sql_query()` function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```python
In [587]: df = pd.DataFrame(np.random.randn(20, 3), columns=list("abc"))
In [588]: df.to_sql("data_chunks", engine, index=False)
Out[588]: 20
```

```python
In [589]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks", engine, chunksize=5):
    ..........: print(chunk)
            .......
...        a     b     c
0  0.092961 -0.674003  1.104153
1  0.092732 -0.156246  0.585167
```

(continues on next page)
You can also run a plain query without creating a DataFrame with `execute()`. This is useful for queries that don’t return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```python
from pandas.io import sql

sql.execute("SELECT * FROM table_name", engine)
sql.execute("INSERT INTO table_name VALUES(?, ?, ?)", engine, params=[("id", 1, 12.2, True)])
```

### Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```python
from sqlalchemy import create_engine

engine = create_engine("postgresql://scott:tiger@localhost:5432/mydatabase")

engine = create_engine("mysql+mysqldb://scott:tiger@localhost/foo")

engine = create_engine("oracle://scott:tiger@127.0.0.1:1521/sidname")

engine = create_engine("mssql+pyodbc://mydsn")

# sqlite:///<nohostname>/<path>
# where <path> is relative:
engine = create_engine("sqlite:///foo.db")
```
# or absolute, starting with a slash:
engine = create_engine("sqlite:///absolute/path/to/foo.db")

For more information see the examples the SQLAlchemy documentation

**Advanced SQLAlchemy queries**

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [590]: import sqlalchemy as sa

In [591]: pd.read_sql(  
    ......:     sa.text("SELECT * FROM data where Col_1=:col1"), engine, params="col1": "X"
    ......: )
    ......:
Out[591]:
index id Date Col_1     Col_2     Col_3  
0     0 26 2010-10-18 00:00:00.000000 X 27.5 1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [592]: metadata = sa.MetaData()

In [593]: data_table = sa.Table(  
    ......:     "data",
    ......:     metadata,
    ......:     sa.Column("index", sa.Integer),
    ......:     sa.Column("Date", sa.DateTime),
    ......:     sa.Column("Col_1", sa.String),
    ......:     sa.Column("Col_2", sa.Float),
    ......:     sa.Column("Col_3", sa.Boolean),
    ......: )
    ......:

In [594]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 is True), engine)
Out[594]:
Empty DataFrame
Columns: [index, Date, Col_1, Col_2, Col_3]
Index: []
```

You can combine SQLAlchemy expressions with parameters passed to `read_sql()` using `sqlalchemy.bindparam()`

```
In [595]: import datetime as dt

In [596]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam("date"))
```

(continues on next page)
In [597]: pd.read_sql(expr, engine, params={"date": dt.datetime(2010, 10, 18)})

Out[597]:

<table>
<thead>
<tr>
<th>index</th>
<th>Date</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2010-10-19</td>
<td>Y</td>
<td>-12.50</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>2010-10-20</td>
<td>Z</td>
<td>5.73</td>
<td>True</td>
</tr>
</tbody>
</table>

Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```python
import sqlite3
con = sqlite3.connect(":memory:"
```

And then issue the following queries:

```python
data.to_sql("data", con)
pd.read_sql_query("SELECT * FROM data", con)
```

### 2.4.17 Google BigQuery

**Warning:** Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package pandas-gbq. You can **pip install pandas-gbq** to get it.

The pandas-gbq package provides functionality to read/write from Google BigQuery.

pandas integrates with this external package. If pandas-gbq is installed, you can use the pandas methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from pandas-gbq.

Full documentation can be found [here](#).

### 2.4.18 Stata format

#### Writing to stata format

The method `to_stata()` will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

In [598]: df = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [599]: df.to_stata("stata.dta")

**Stata** data files have limited data type support; only strings with 244 or fewer characters, int8, int16, int32, float32 and float64 can be stored in .dta files. Additionally, Stata reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in Stata for a particular data type will retype the variable to the next larger size. For example, int8 values are restricted to lie between -127 and 100 in Stata, and so variables

2.4. IO tools (text, CSV, HDF5, ...)}
with values above 100 will trigger a conversion to int16. nan values in floating points data types are stored as the basic missing data type (. in Stata).

**Note:** It is not possible to export missing data values for integer data types.

The Stata writer gracefully handles other data types including int64, bool, uint8, uint16, uint32 by casting to the smallest supported type that can represent the data. For example, data with a type of uint8 will be cast to int8 if all values are less than 100 (the upper bound for non-missing int8 data in Stata), or, if values are outside of this range, the variable is cast to int16.

**Warning:** Conversion from int64 to float64 may result in a loss of precision if int64 values are larger than $2^{**53}$.

**Warning:** StataWriter and to_stata() only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write Stata dta files with strings longer than 244 characters raises a ValueError.

### Reading from Stata format

The top-level function read_stata will read a dta file and return either a DataFrame or a StataReader that can be used to read the file incrementally.

```python
In [600]: pd.read_stata("stata.dta")
Out[600]:
     index   A       B
0       0  0.608228  1.064810
1       1 -0.780506 -2.736887
2       2  0.143539  1.170191
3       3 -1.573076  0.075792
4       4 -1.722223 -0.774650
5       5  0.803627  0.221665
6       6  0.584637  0.147264
7       7  1.057825 -0.284136
8       8  0.912395  1.552808
9       9  0.189376 -0.109830
```

Specifying a chunksize yields a StataReader instance that can be used to read chunksize lines from the file at a time. The StataReader object can be used as an iterator.

```python
In [601]: with pd.read_stata("stata.dta", chunksize=3) as reader:
   .....:     for df in reader:
   .....:       print(df.shape)
   .....: (3, 3)
   (3, 3)
   (3, 3)
   (1, 3)
```

For more fine-grained control, use iterator=True and specify chunksize with each call to read().
Currently the index is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a Categorical variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If False (the default), missing values are represented as `np.nan`. If True, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have object data type.

**Note:** `read_stata()` and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

**Note:** Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

### Categorical data

Categorical data can be exported to Stata data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. Stata does not have an explicit equivalent to a Categorical and information about whether the variable is ordered is lost when exporting.

**Warning:** Stata only supports string value labels, and so `str` is called on the categories when exporting data. Exporting Categorical variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from Stata data files as Categorical variables using the keyword argument `convert_categoricals` (True by default). The keyword argument `order_categoricals` (True by default) determines whether imported Categorical variables are ordered.

**Note:** When importing categorical data, the values of the variables in the Stata data file are not preserved since Categorical variables always use integer data types between -1 and n-1 where n is the number of categories. If the original values in the Stata data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original Stata data values and the category codes of imported Categorical variables: missing values are assigned code -1, and the smallest original value is assigned 0, the second smallest is assigned 1 and so on until the largest original value is assigned the code n-1.

**Note:** Stata supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a Categorical with string categories for the values that are labeled and numeric categories for values with no label.
2.4.19 SAS formats

The top-level function `read_sas()` can read (but not write) SAS XPORT (.xpt) and (since v0.18.0) SAS7BDAT (.sas7bdat) format files.

SAS files only contain two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates. By default the whole file is read and returned as a `DataFrame`.

Specify a `chunksize` or use `iterator=True` to obtain reader objects (`XportReader` or `SAS7BDATReader`) for incrementally reading the file. The reader objects also have attributes that contain additional information about the file and its variables.

Read a SAS7BDAT file:

```python
df = pd.read_sas("sas_data.sas7bdat")
```

Obtain an iterator and read an XPORT file 100,000 lines at a time:

```python
def do_something(chunk):
    pass

with pd.read_sas("sas_xport.xpt", chunk=100000) as rdr:
    for chunk in rdr:
        do_something(chunk)
```

The specification for the xport file format is available from the SAS web site.

No official documentation is available for the SAS7BDAT format.

2.4.20 SPSS formats

New in version 0.25.0.

The top-level function `read_spss()` can read (but not write) SPSS SAV (.sav) and ZSAV (.zsav) format files.

SPSS files contain column names. By default the whole file is read, categorical columns are converted into `pd.Categorical`, and a DataFrame with all columns is returned.

Specify the `usecols` parameter to obtain a subset of columns. Specify `convert_categoricals=False` to avoid converting categorical columns into `pd.Categorical`.

Read an SPSS file:

```python
df = pd.read_spss("spss_data.sav")
```

Extract a subset of columns contained in `usecols` from an SPSS file and avoid converting categorical columns into `pd.Categorical`:

```python
df = pd.read_spss("spss_data.sav",
    usecols=["foo", "bar"],
    convert_categoricals=False,
)
```

More information about the SAV and ZSAV file formats is available here.
2.4.21 Other file formats

pandas itself only supports IO with a limited set of file formats that map cleanly to its tabular data model. For reading and writing other file formats into and from pandas, we recommend these packages from the broader community.

netCDF

xarray provides data structures inspired by the pandas DataFrame for working with multi-dimensional datasets, with a focus on the netCDF file format and easy conversion to and from pandas.

2.4.22 Performance considerations

This is an informal comparison of various IO methods, using pandas 0.24.2. Timings are machine dependent and small differences should be ignored.

```python
In [1]: sz = 1000000

In [3]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A 1000000 non-null float64
B 1000000 non-null int64
dtypes: float64(1), int64(1)
memory usage: 15.3 MB
```

The following test functions will be used below to compare the performance of several IO methods:

```python
import numpy as np
import os
sz = 1000000

sz = 1000000
np.random.seed(42)

def test_sql_write(df):
    if os.path.exists("test.sql"):
        os.remove("test.sql")
    sql_db = sqlite3.connect("test.sql")
    df.to_sql(name="test_table", con=sql_db)
    sql_db.close()

def test_sql_read():
    sql_db = sqlite3.connect("test.sql")
    pd.read_sql_query("select * from test_table", sql_db)
```

(continues on next page)
sql_db.close()

def test_hdf_fixed_write(df):
    df.to_hdf("test_fixed.hdf", "test", mode="w")

def test_hdf_fixed_read():
    pd.read_hdf("test_fixed.hdf", "test")

def test_hdf_fixed_write_compress(df):
    df.to_hdf("test_fixed_compress.hdf", "test", mode="w", complib="blosc")

def test_hdf_fixed_read_compress():
    pd.read_hdf("test_fixed_compress.hdf", "test")

def test_hdf_table_write(df):
    df.to_hdf("test_table.hdf", "test", mode="w", format="table")

def test_hdf_table_read():
    pd.read_hdf("test_table.hdf", "test")

def test_hdf_table_write_compress(df):
    df.to_hdf("test_table_compress.hdf", "test", mode="w", complib="blosc", format="table")

def test_hdf_table_read_compress():
    pd.read_hdf("test_table_compress.hdf", "test")

def test_csv_write(df):
    df.to_csv("test.csv", mode="w")

def test_csv_read():
    pd.read_csv("test.csv", index_col=0)

def test_feather_write(df):
    df.to_feather("test.feather")

def test_feather_read():
    pd.read_feather("test.feather")
def test_pickle_write(df):
    df.to_pickle("test.pkl")

def test_pickle_read():
    pd.read_pickle("test.pkl")

def test_pickle_write_compress(df):
    df.to_pickle("test.pkl.compress", compression="xz")

def test_pickle_read_compress():
    pd.read_pickle("test.pkl.compress", compression="xz")

def test_parquet_write(df):
    df.to_parquet("test.parquet")

def test_parquet_read():
    pd.read_parquet("test.parquet")

When writing, the top three functions in terms of speed are test_feather_write, test_hdf_fixed_write and test_hdf_fixed_write_compress.
When reading, the top three functions in terms of speed are `test_feather_read`, `test_pickle_read` and `test_hdf_fixed_read`.

The files `test.pkl.compress`, `test.parquet` and `test.feather` took the least space on disk (in bytes).
2.5 Indexing and selecting data

The axis labeling information in pandas objects serves many purposes:

- Identifies data (i.e. provides metadata) using known indicators, important for analysis, visualization, and interactive console display.
- Enables automatic and explicit data alignment.
- Allows intuitive getting and setting of subsets of the data set.

In this section, we will focus on the final point: namely, how to slice, dice, and generally get and set subsets of pandas objects. The primary focus will be on Series and DataFrame as they have received more development attention in this area.

**Note:** The Python and NumPy indexing operators [] and attribute operator . provide quick and easy access to pandas data structures across a wide range of use cases. This makes interactive work intuitive, as there's little new to learn if you already know how to deal with Python dictionaries and NumPy arrays. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommended that you take advantage of the optimized pandas data access methods exposed in this chapter.

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called chained assignment and should be avoided. See *Returning a View versus Copy*.

See the MultiIndex / Advanced Indexing for MultiIndex and more advanced indexing documentation.

See the cookbook for some advanced strategies.

2.5.1 Different choices for indexing

Object selection has had a number of user-requested additions in order to support more explicit location based indexing. pandas now supports three types of multi-axis indexing.

- .loc is primarily label based, but may also be used with a boolean array. .loc will raise `KeyError` when the items are not found. Allowed inputs are:
  - A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index.).
  - A list or array of labels ['a', 'b', 'c'].
  - A slice object with labels 'a': 'f' (Note that contrary to usual Python slices, both the start and the stop are included, when present in the index! See *Slicing with labels and Endpoints are inclusive.*).
  - A boolean array (any NA values will be treated as False).
  - A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

See more at Selection by Label.

- .iloc is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array. .iloc will raise `IndexError` if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy slice semantics). Allowed inputs are:
  - An integer e.g. 5.
- A list or array of integers [4, 3, 0].
- A slice object with ints 1:7.
- A boolean array (any NA values will be treated as False).
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

See more at Selection by Position, Advanced Indexing and Advanced Hierarchical.

- .loc, .iloc, and also [] indexing can accept a callable as indexer. See more at Selection By Callable.

Getting values from an object with multi-axes selection uses the following notation (using .loc as an example, but the following applies to .iloc as well). Any of the axes accessors may be the null slice ::. Axes left out of the specification are assumed to be :, e.g. p.loc['a'] is equivalent to p.loc['a', :, :].

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Indexers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>s.loc[indexer]</td>
</tr>
<tr>
<td>DataFrame</td>
<td>df.loc[row_indexer, column_indexer]</td>
</tr>
</tbody>
</table>

### 2.5.2 Basics

As mentioned when introducing the data structures in the last section, the primary function of indexing with [] (a.k.a. __getitem__ for those familiar with implementing class behavior in Python) is selecting out lower-dimensional slices. The following table shows return type values when indexing pandas objects with []:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Selection</th>
<th>Return Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>series[label]</td>
<td>scalar value</td>
</tr>
<tr>
<td>DataFrame</td>
<td>frame[colname]</td>
<td>Series corresponding to colname</td>
</tr>
</tbody>
</table>

Here we construct a simple time series data set to use for illustrating the indexing functionality:

```
In [1]: dates = pd.date_range('1/1/2000', periods=8)
In [2]: df = pd.DataFrame(np.random.randn(8, 4),          ...
                   index=dates, columns=['A', 'B', 'C', 'D'])
In [3]: df
Out[3]:          A         B         C         D
2000-01-01 -0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705 -1.157892 -1.344312  0.844885
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
```

**Note:** None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using []:
In [4]: s = df['A']

In [5]: s[dates[5]]
Out[5]: -0.67369070883706

You can pass a list of columns to [] to select columns in that order. If a column is not contained in the DataFrame, an exception will be raised. Multiple columns can also be set in this manner:

In [6]: df
Out[6]:
   A   B   C   D
0  2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
1  2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2  2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
3  2000-01-04  0.721555 -0.706771 -1.039575  0.271860
4  2000-01-05 -0.424972  0.567020  0.276232 -1.087401
5  2000-01-06 -0.673690  0.113648 -1.478427  0.524988
6  2000-01-07  0.404705  0.577046 -1.715002 -1.039268
7  2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

In [7]: df[['B', 'A']] = df[['A', 'B']]

In [8]: df
Out[8]:
   A  B   C   D
0  2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
1  2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2  2000-01-03 -2.104569 -0.861849  0.494929  1.071804
3  2000-01-04 -0.706771 -0.706771 -1.039575  0.271860
4  2000-01-05  0.567020 -0.424972  0.276232 -1.087401
5  2000-01-06  0.113648 -0.673690 -1.478427  0.524988
6  2000-01-07  0.577046  0.404705 -1.715002 -1.039268
7  2000-01-08 -1.157892 -0.370647 -1.344312  0.844885

You may find this useful for applying a transform (in-place) to a subset of the columns.

Warning: pandas aligns all AXES when setting Series and DataFrame from .loc, and .iloc.

This will not modify df because the column alignment is before value assignment.

In [9]: df[['A', 'B']]  
Out[9]:
   A   B
0  2000-01-01 -0.282863  0.469112
1  2000-01-02 -0.173215  1.212112
2  2000-01-03 -2.104569 -0.861849
3  2000-01-04 -0.706771  0.721555
4  2000-01-05  0.567020 -0.424972
5  2000-01-06  0.113648 -0.673690
6  2000-01-07  0.577046  0.404705
7  2000-01-08 -1.157892 -0.370647

In [10]: df.loc[::, ['B', 'A']] = df[['A', 'B']]

2.5. Indexing and selecting data 413
2.5.3 Attribute access

You may access an index on a Series or column on a DataFrame directly as an attribute:

In [14]: sa = pd.Series([1, 2, 3], index=list('abc'))

In [15]: dfa = df.copy()

In [16]: sa.b
Out[16]: 2

In [17]: dfa.A
Out[17]:
2000-01-01    0.469112
2000-01-02    1.212112
2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
2000-01-06   -0.673690
2000-01-07    0.404705
2000-01-08   -0.370647
Freq: D, Name: A, dtype: float64
In [18]: sa.a = 5

In [19]: sa
Out[19]:
   a    5
   b    2
   c    3
dtype: int64

In [20]: dfa.A = list(range(len(dfa.index)))  # ok if A already exists

In [21]: dfa
Out[21]:
     A  B   C     D
 2000-01-01  0 -0.282863 -1.509059 -1.135632
 2000-01-02  1 -0.173215  0.119209 -1.044236
 2000-01-03  2 -2.104569 -0.494929  1.071804
 2000-01-04  3 -0.706771 -1.039575  0.271860
 2000-01-05  4  0.567020  0.276232 -1.087401
 2000-01-06  5  0.113648 -1.478427  0.524988
 2000-01-07  6  0.577046 -1.715002 -1.039268
 2000-01-08  7 -1.157892 -1.344312  0.844885

In [22]: dfa['A'] = list(range(len(dfa.index)))  # use this form to create a new column

In [23]: dfa
Out[23]:
     A  B   C     D
 2000-01-01  0 -0.282863 -1.509059 -1.135632
 2000-01-02  1 -0.173215  0.119209 -1.044236
 2000-01-03  2 -2.104569 -0.494929  1.071804
 2000-01-04  3 -0.706771 -1.039575  0.271860
 2000-01-05  4  0.567020  0.276232 -1.087401
 2000-01-06  5  0.113648 -1.478427  0.524988
 2000-01-07  6  0.577046 -1.715002 -1.039268
 2000-01-08  7 -1.157892 -1.344312  0.844885

Warning:

- You can use this access only if the index element is a valid Python identifier, e.g. s.1 is not allowed. See here for an explanation of valid identifiers.
- The attribute will not be available if it conflicts with an existing method name, e.g. s.min is not allowed, but s['min'] is possible.
- Similarly, the attribute will not be available if it conflicts with any of the following list: index, major_axis, minor_axis, items.
- In any of these cases, standard indexing will still work, e.g. s['1'], s['min'], and s['index'] will access the corresponding element or column.

If you are using the IPython environment, you may also use tab-completion to see these accessible attributes.
You can also assign a dict to a row of a DataFrame:

### 2.5. Indexing and selecting data
You can use attribute access to modify an existing element of a Series or column of a DataFrame, but be careful; if you try to use attribute access to create a new column, it creates a new attribute rather than a new column. In 0.21.0 and later, this will raise a UserWarning:

```python
In [1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In [2]: df.two = [4, 5, 6]
UserWarning: Pandas doesn't allow Series to be assigned into nonexistent columns - see...
  → https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute_access
In [3]: df
Out[3]:
   one
0  1.0
1  2.0
2  3.0
```

### 2.5.4 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the *Selection by Position* section detailing the `.iloc` method. For now, we explain the semantics of slicing using the `[]` operator.

With Series, the syntax works exactly as with an ndarray, returning a slice of the values and the corresponding labels:

```python
In [27]: s[:5]
Out[27]:
   2000-01-01  0.469112
   2000-01-02  1.212112
   2000-01-03 -0.861849
   2000-01-04  0.721555
   2000-01-05 -0.424972
Freq: D, Name: A, dtype: float64

In [28]: s[::2]
Out[28]:
   2000-01-01  0.469112
   2000-01-03 -0.861849
   2000-01-05 -0.424972
   2000-01-07  0.404705
Freq: 2D, Name: A, dtype: float64

In [29]: s[::-1]
Out[29]:
   (continues on next page)
```
2000-01-08  -0.370647
2000-01-07   0.404705
2000-01-06  -0.673690
2000-01-05  -0.424972
2000-01-04   0.721555
2000-01-03  -0.861849
2000-01-02   1.212112
2000-01-01   0.469112
Freq: -1D, Name: A, dtype: float64

Note that setting works as well:

```
In [30]: s2 = s.copy()
In [31]: s2[:5] = 0
```
```
In [32]: s2
Out[32]:
2000-01-01  0.000000
2000-01-02  0.000000
2000-01-03  0.000000
2000-01-04  0.000000
2000-01-05  0.000000
2000-01-06 -0.673690
2000-01-07  0.404705
2000-01-08 -0.370647
Freq: D, Name: A, dtype: float64
```

With DataFrame, slicing inside of `[]` slices the rows. This is provided largely as a convenience since it is such a common operation.

```
In [33]: df[:3]
Out[33]:
          A       B       C       D
2000-01-01 0.469112 -0.282863 -1.509059 -1.135632
2000-01-02 1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
```
```
In [34]: df[::-1]
Out[34]:
          A       B       C       D
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
2000-01-07 0.404705  0.577046 -1.715002 -1.039268
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-04 0.721555 -0.706771 -1.039575  0.271860
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-02 1.212112 -0.173215  0.119209 -1.044236
2000-01-01 0.469112 -0.282863 -1.509059 -1.135632
```

2.5. Indexing and selecting data
2.5.5 Selection by label

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called chained assignment and should be avoided. See *Returning a View versus Copy.*

**Warning:**

.loc is strict when you present slicers that are not compatible (or convertible) with the index type. For example using integers in a DatetimeIndex. These will raise a TypeError.

```
In [35]: dfl = pd.DataFrame(np.random.randn(5, 4),
....:  columns=list('ABCD'),
....:  index=pd.date_range('20130101', periods=5))
....:
In [36]: dfl
Out[36]:
   A      B      C      D
2013-01-01  1.0758  1.6436 -1.4694 -1.4694
2013-01-02 -0.3570 -0.6746 -1.7769 -0.9689
2013-01-03 -1.2945  0.4137  0.2767 -0.4720
2013-01-04 -0.0140 -0.3625 -0.0062 -0.9231
2013-01-05  0.8957  0.8052 -1.2064  2.5656
```

```
In [4]: dfl.loc[2:3]
TypeError: cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'> with these indexers [2] of <type 'int'>
```

String likes in slicing can be convertible to the type of the index and lead to natural slicing.

```
In [37]: dfl.loc['20130102':'20130104']
Out[37]:
   A      B      C      D
2013-01-02 -0.3570 -0.6746 -1.7769 -0.9689
2013-01-03 -1.2945  0.4137  0.2767 -0.4720
2013-01-04 -0.0140 -0.3625 -0.0062 -0.9231
```

**Warning:** Changed in version 1.0.0.

pandas will raise a **KeyError** if indexing with a list with missing labels. See *list-like Using loc with missing keys in a list is Deprecated.*

pandas provides a suite of methods in order to have purely label based indexing. This is a strict inclusion based protocol. Every label asked for must be in the index, or a **KeyError** will be raised. When slicing, both the start bound **AND** the stop bound are included, if present in the index. Integers are valid labels, but they refer to the label **and not** the position.

The .loc attribute is the primary access method. The following are valid inputs:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is **not** an integer position along the index.).
• A list or array of labels ['a', 'b', 'c'].
• A slice object with labels 'a':'f' (Note that contrary to usual Python slices, both the start and the stop are included, when present in the index! See Slicing with labels.
• A boolean array.
• A callable, see Selection By Callable.

In [38]: s1 = pd.Series(np.random.randn(6), index=list('abcdef'))
In [39]: s1
Out[39]:
   a   1.431256
   b   1.340309
   c  -1.170299
   d  -0.226169
   e   0.410835
   f   0.813850
 dtype: float64

In [40]: s1.loc['c:']
Out[40]:
   c  -1.170299
   d  -0.226169
   e   0.410835
   f   0.813850
 dtype: float64

In [41]: s1.loc['b']
Out[41]: 1.3403088497993827

Note that setting works as well:

In [42]: s1.loc['c'] = 0
In [43]: s1
Out[43]:
   a   1.431256
   b   1.340309
   c   0.000000
   d   0.000000
   e   0.000000
   f   0.000000
 dtype: float64

With a DataFrame:

In [44]: df1 = pd.DataFrame(np.random.randn(6, 4),
   index=list('abcdef'),
   columns=list('ABCD'))
In [45]: df1
Out[45]:
(continues on next page)
Accessing via label slices:

```python
In [47]: df1.loc['d', 'A':'C']
Out[47]:
   A     B     C
d 0.97446 -2.00675 -0.41000
```

For getting a cross section using a label (equivalent to `df1.xs('a')):

```python
In [48]: df1.loc['a']
Out[48]:
   A    B    C    D
Name: a, dtype: float64
```

For getting values with a boolean array:

```python
In [49]: df1.loc['a'] > 0
Out[49]:
   A  True
   B  False
   C  False
   D  False
Name: a, dtype: bool
```

```python
In [50]: df1.loc[:, df1.loc['a'] > 0]
Out[50]:
   A
a 0.132003
b 1.130127
c 1.024180
d 0.974466
e 0.545952
f -1.281247
```
NA values in a boolean array propagate as False:

Changed in version 1.0.2.

```python
In [51]: mask = pd.array([True, False, True, False, pd.NA, False], dtype="boolean")
```

```python
In [52]: mask
Out[52]: <BooleanArray>
[True, False, True, False, <NA>, False]
Length: 6, dtype: boolean
```

```python
In [53]: df1[mask]
Out[53]:
        A     B     C     D
a  0.132003 -0.827317 -0.076467 -1.187678
c  1.024180  0.569605  0.875906 -2.211372
```

For getting a value explicitly:

```python
In [54]: df1.loc['a', 'A']
Out[54]: 0.13200317033032932
```

### Slicing with labels

When using `.loc` with slices, if both the start and the stop labels are present in the index, then elements *located* between the two (including them) are returned:

```python
In [55]: s = pd.Series(list('abcde'), index=[0, 3, 2, 5, 4])
```

```python
In [56]: s.loc[3:5]
Out[56]:
3  b
2  c
dtype: object
```

If at least one of the two is absent, but the index is sorted, and can be compared against start and stop labels, then slicing will still work as expected, by selecting labels which *rank* between the two:

```python
In [57]: s.sort_index()
Out[57]:
0  a
2  c
3  b
4  e
5  d
dtype: object
```

```python
In [58]: s.sort_index().loc[1:6]
Out[58]:
2  c
```

(continues on next page)
However, if at least one of the two is absent and the index is not sorted, an error will be raised (since doing otherwise would be computationally expensive, as well as potentially ambiguous for mixed type indexes). For instance, in the above example, `s.loc[1:6]` would raise `KeyError`.

For the rationale behind this behavior, see *Endpoints are inclusive.*

```python
In [59]: s = pd.Series(list('abcdef'), index=[0, 3, 2, 5, 4, 2])

In [60]: s.loc[3:5]
Out[60]:
3  b
2  c
5  d
dtype: object
```

Also, if the index has duplicate labels and either the start or the stop label is duplicated, an error will be raised. For instance, in the above example, `s.loc[2:5]` would raise a `KeyError`.

For more information about duplicate labels, see *Duplicate Labels.*

### 2.5.6 Selection by position

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called chained assignment and should be avoided. See *Returning a View versus Copy.*

pandas provides a suite of methods in order to get purely integer based indexing. The semantics follow closely Python and NumPy slicing. These are 0-based indexing. When slicing, the start bound is included, while the upper bound is excluded. Trying to use a non-integer, even a valid label will raise an `IndexError`.

The `.iloc` attribute is the primary access method. The following are valid inputs:

- An integer e.g. 5.
- A list or array of integers [4, 3, 0].
- A slice object with ints 1:7.
- A boolean array.
- A callable, see *Selection By Callable.*

```python
In [61]: s = pd.Series(np.random.randn(5), index=list(range(0, 10, 2)))

In [62]: s
Out[62]:
0  0.695775
2  0.695775
4  0.695775
6  0.695775
8  0.695775
```
In [63]: s1.iloc[:3]
Out[63]:
0  0.695775
2  0.341734
4  0.959726
dtype: float64

In [64]: s1.iloc[3]
Out[64]:
-1.110336102891167

Note that setting works as well:

In [65]: s1.iloc[:3] = 0

In [66]: s1
Out[66]:
   0   2   4   6
0  0.000000
2  0.000000
4  0.000000
6 -1.110336
8 -0.619976
dtype: float64

With a DataFrame:

In [67]: df1 = pd.DataFrame(np.random.randn(6, 4),
       index=list(range(0, 12, 2)),
       columns=list(range(0, 8, 2)))

In [68]: df1
Out[68]:
     0   2   4   6
0  0.149748 -0.732339 0.687738 0.176444
2  0.403310 -0.154951 0.301624 -2.179861
4 -1.369849 -0.954208 1.462696 -1.743161
6 -0.826591 -0.345352 1.314232  0.690579
8  0.995761  2.396780  0.014871  3.357427
10 -0.317441 -1.236269  0.896171 -0.487602

Select via integer slicing:

In [69]: df1.iloc[:3]
Out[69]:
     0   2   4   6
0  0.149748 -0.732339 0.687738 0.176444
2  0.403310 -0.154951 0.301624 -2.179861
4 -1.369849 -0.954208 1.462696 -1.743161

In [70]: df1.iloc[1:5, 2:4]

(continues on next page)
Select via integer list:

```
In [71]: df1.iloc[[1, 3, 5], [1, 3]]
Out[71]:
     2       6
0 -0.154951 -2.179861
4 -0.345352  0.690579
8 -1.236269 -0.487602
```

```
In [72]: df1.iloc[1:3, :]
Out[72]:
      2  4  6
0 0.403310 0.687738
2 -0.154951 0.301624 -2.179861
4 -0.954208 1.462696 -1.743161
```

```
In [73]: df1.iloc[:, 1:3]
Out[73]:
      2  4
0 -0.732339 0.687738
2 -0.154951 0.301624
4 -0.954208 1.462696
6 -0.345352 1.314232
8  2.396780 0.014871
10 -1.236269 0.896171
```

# this is also equivalent to `df1.iat[1, 1]`

```
In [74]: df1.iloc[1, 1]
Out[74]:
-0.1549507744249032
```

For getting a cross section using an integer position (equiv to df.xs(1)):

```
In [75]: df1.iloc[1]
Out[75]:
      0       2       4       6
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/NumPy.

```
# these are allowed in Python/NumPy.
In [76]: x = list('abcdef')

In [77]: x
```
Out[77]: ['a', 'b', 'c', 'd', 'e', 'f']

In [78]: x[4:10]
Out[78]: ['e', 'f']

In [79]: x[8:10]
Out[79]: []

In [80]: s = pd.Series(x)

In [81]: s
Out[81]:
0   a
1   b
2   c
3   d
4   e
5   f
dtype: object

In [82]: s.iloc[4:10]
Out[82]:
4   e
5   f
dtype: object

In [83]: s.iloc[8:10]
Out[83]: Series([], dtype: object)

Note that using slices that go out of bounds can result in an empty axis (e.g. an empty DataFrame being returned).

In [84]: dfl = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [85]: dfl
Out[85]:
    A       B
0 -0.082240 -2.182937
1  0.380396  0.084844
2  0.432390  1.519970
3 -0.493662  0.600178
4  0.274230  0.132885

In [86]: dfl.iloc[:, 2:3]
Out[86]:
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [87]: dfl.iloc[:, 1:3]
Out[87]:
    B
0 -2.182937

2.5. Indexing and selecting data
A single indexer that is out of bounds will raise an `IndexError`. A list of indexers where any element is out of bounds will raise an `IndexError`.

```python
>>> dfl.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds
```

```python
>>> dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

### 2.5.7 Selection by callable

`.loc`, `.iloc`, and also `[]` indexing can accept a callable as indexer. The callable must be a function with one argument (the calling Series or DataFrame) that returns valid output for indexing.

```python
In [89]: df1 = pd.DataFrame(np.random.randn(6, 4),
....:                     index=list('abcdef'),
....:                     columns=list('ABCD'))
....:
```

```python
In [90]: df1
Out[90]:
   A    B    C    D
a -0.023688 2.410179 1.450520 0.206053
b -0.251905 -2.213588 1.063327 1.266143
c  0.299368 -0.863838 0.408204 -1.048089
d -0.025747 -0.988387 0.094055 1.262731
e  1.289997  0.082423 -0.055758 0.536580
f -0.489682  0.369374 -0.034571 -2.484478
```

```python
In [91]: df1.loc[lambda df: df['A'] > 0, :]
Out[91]:
   A    B    C    D
  c  0.299368 -0.863838 0.408204 -1.048089
e  1.289997  0.082423 -0.055758 0.536580
```

```python
In [92]: df1.loc[:, lambda df: ['A', 'B']]
Out[92]:
   A    B
a -0.023688 2.410179
b -0.251905 -2.213588
c  0.299368 -0.863838
```

(continues on next page)
In [93]: df1.iloc[:, lambda df: [0, 1]]
Out[93]:
   A    B
a -0.023688  2.410179
b -0.251905 -2.213588
c  0.299368  -0.863838
d -0.025747 -0.988387
e  1.289997  0.082423
f -0.489682  0.369374

In [94]: df1[lambda df: df.columns[0]]
Out[94]:
a -0.023688
b -0.251905
c  0.299368
d -0.025747
e  1.289997
f -0.489682
Name: A, dtype: float64

You can use callable indexing in Series.

In [95]: df1['A'].loc[lambda s: s > 0]
Out[95]:
c  0.299368
e  1.289997
Name: A, dtype: float64

Using these methods/indexers, you can chain data selection operations without using a temporary variable.

In [96]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [97]: (bb.groupby(['year', 'team']).sum()
   ....:   .loc[lambda df: df['r'] > 100])
Out[97]:
<table>
<thead>
<tr>
<th>year</th>
<th>team</th>
<th>stint</th>
<th>g</th>
<th>ab</th>
<th>r</th>
<th>h</th>
<th>X2b</th>
<th>X3b</th>
<th>hr</th>
<th>rbi</th>
<th>sb</th>
<th>cs</th>
<th>bb</th>
<th>so</th>
<th>ibb</th>
<th>hbp</th>
<th>sh</th>
<th>sf</th>
<th>gidp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2007</td>
<td>CIN</td>
<td>6</td>
<td>379</td>
<td>745</td>
<td>101</td>
<td>203</td>
<td>35</td>
<td>2</td>
<td>36</td>
<td>125.0</td>
<td>10.0</td>
<td>1.0</td>
<td>105</td>
<td>127.0</td>
<td>14.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td>15.0</td>
<td>18.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DET</td>
<td>5</td>
<td>301</td>
<td>1062</td>
<td>162</td>
<td>283</td>
<td>54</td>
<td>4</td>
<td>37</td>
<td>144.0</td>
<td>24.0</td>
<td>7.0</td>
<td>97</td>
<td>176.0</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HOU</td>
<td>4</td>
<td>311</td>
<td>926</td>
<td>109</td>
<td>218</td>
<td>47</td>
<td>6</td>
<td>14</td>
<td>77.0</td>
<td>10.0</td>
<td>4.0</td>
<td>60</td>
<td>212.0</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAN</td>
<td>11</td>
<td>413</td>
<td>1021</td>
<td>153</td>
<td>293</td>
<td>61</td>
<td>3</td>
<td>36</td>
<td>154.0</td>
<td>7.0</td>
<td>5.0</td>
<td>114</td>
<td>141.0</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.0</td>
<td>3.0</td>
<td>8.0</td>
<td>29.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
Combining positional and label-based indexing

If you wish to get the 0th and the 2nd elements from the index in the ‘A’ column, you can do:

```python
In [98]: dfd = pd.DataFrame({'A': [1, 2, 3],
                          'B': [4, 5, 6],
                          index=list('abc'))

In [99]: dfd
Out[99]:
          A  B
a  1  4
b  2  5
c  3  6
```

This can also be expressed using `.iloc`, by explicitly getting locations on the indexers, and using *positional* indexing to select things.

```python
In [100]: dfd.iloc[[0, 2], dfd.columns.get_loc('A')]
Out[100]:
   A
a  1
c  3
Name: A, dtype: int64
```

For getting *multiple* indexers, using `.get_indexer`:

```python
In [101]: dfd.iloc[[0, 2], dfd.columns.get_indexer(['A', 'B'])]
Out[101]:
   A  B
a  1  4
c  3  6
```
### 2.5.9 Indexing with list with missing labels is deprecated

**Warning:** Changed in version 1.0.0.

Using `.loc` or `[]` with a list with one or more missing labels will no longer reindex, in favor of `.reindex`.

In prior versions, using `.loc[list-of-labels]` would work as long as *at least 1* of the keys was found (otherwise it would raise a `KeyError`). This behavior was changed and will now raise a `KeyError` if at least one label is missing. The recommended alternative is to use `.reindex()`.

For example.

```python
In [103]: s = pd.Series([1, 2, 3])

In [104]: s
Out[104]:
0    1
1    2
2    3
dtype: int64
```

Selection with all keys found is unchanged.

```python
In [105]: s.loc[[1, 2]]
Out[105]:
1    2
2    3
dtype: int64
```

Previous behavior

```python
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Current behavior

```python
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc with any non-matching elements will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
```

```python
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```
Reindexing

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`. See also the section on reindexing.

```
In [106]: s.reindex([1, 2, 3])
Out[106]:
   1  2.0
   2  3.0
   3  NaN
dtype: float64
```

Alternatively, if you want to select only valid keys, the following is idiomatic and efficient; it is guaranteed to preserve the dtype of the selection.

```
In [107]: labels = [1, 2, 3]
In [108]: s.loc[s.index.intersection(labels)]
Out[108]:
   1  2
   2  3
dtype: int64
```

Having a duplicated index will raise for a `.reindex()`:

```
In [109]: s = pd.Series(np.arange(4), index=['a', 'a', 'b', 'c'])
In [110]: labels = ['c', 'd']
In [17]: s.reindex(labels)
ValueError: cannot reindex on an axis with duplicate labels
```

Generally, you can intersect the desired labels with the current axis, and then reindex.

```
In [111]: s.loc[s.index.intersection(labels)].reindex(labels)
Out[111]:
   c  3.0
d  NaN
dtype: float64
```

However, this would still raise if your resulting index is duplicated.

```
In [41]: labels = ['a', 'd']
In [42]: s.loc[s.index.intersection(labels)].reindex(labels)
ValueError: cannot reindex on an axis with duplicate labels
```
2.5.10 Selecting random samples

A random selection of rows or columns from a Series or DataFrame with the `sample()` method. The method will sample rows by default, and accepts a specific number of rows/columns to return, or a fraction of rows.

```
In [112]: s = pd.Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1 row.
In [113]: s.sample()
Out[113]:
4  4
dtype: int64

# One may specify either a number of rows:
In [114]: s.sample(n=3)
Out[114]:
0  0
4  4
1  1
dtype: int64

# Or a fraction of the rows:
In [115]: s.sample(frac=0.5)
Out[115]:
5  5
3  3
1  1
dtype: int64
```

By default, `sample` will return each row at most once, but one can also sample with replacement using the `replace` option:

```
In [116]: s = pd.Series([0, 1, 2, 3, 4, 5])

# Without replacement (default):
In [117]: s.sample(n=6, replace=False)
Out[117]:
0  0
1  1
5  5
3  3
2  2
4  4
dtype: int64

# With replacement:
In [118]: s.sample(n=6, replace=True)
Out[118]:
0  0
4  4
3  3
2  2
4  4
4  4
```

(continues on next page)
By default, each row has an equal probability of being selected, but if you want rows to have different probabilities, you can pass the `sample` function sampling weights as `weights`. These weights can be a list, a NumPy array, or a Series, but they must be of the same length as the object you are sampling. Missing values will be treated as a weight of zero, and inf values are not allowed. If weights do not sum to 1, they will be re-normalized by dividing all weights by the sum of the weights. For example:

```
In [119]: s = pd.Series([0, 1, 2, 3, 4, 5])
In [120]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]
In [121]: s.sample(n=3, weights=example_weights)
Out[121]:
5 5
4 4
3 3
dtype: int64
```

# Weights will be re-normalized automatically
```
In [122]: example_weights2 = [0.5, 0, 0, 0, 0, 0]
In [123]: s.sample(n=1, weights=example_weights2)
Out[123]:
0 0
dtype: int64
```

When applied to a DataFrame, you can use a column of the DataFrame as sampling weights (provided you are sampling rows and not columns) by simply passing the name of the column as a string.

```
In [124]: df2 = pd.DataFrame({'col1': [9, 8, 7, 6],
                      'weight_column': [0.5, 0.4, 0.1, 0])
In [125]: df2.sample(n=3, weights='weight_column')
Out[125]:
col1  weight_column
1 8 0.4
0 9 0.5
2 7 0.1
```

`sample` also allows users to sample columns instead of rows using the `axis` argument.

```
In [126]: df3 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})
In [127]: df3.sample(n=1, axis=1)
Out[127]:
col1
0 1
1 2
2 3
```

Finally, one can also set a seed for `sample`’s random number generator using the `random_state` argument, which will...
accept either an integer (as a seed) or a NumPy RandomState object.

```
In [128]: df4 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

# With a given seed, the sample will always draw the same rows.
In [129]: df4.sample(n=2, random_state=2)
Out[129]:
   col1 col2
0    2    3
1    1    2

In [130]: df4.sample(n=2, random_state=2)
Out[130]:
   col1 col2
0    2    3
1    1    2
```

### 2.5.11 Setting with enlargement

The `.loc/[]` operations can perform enlargement when setting a non-existent key for that axis. In the Series case this is effectively an appending operation.

```
In [131]: se = pd.Series([1, 2, 3])

In [132]: se
Out[132]:
0     1
1     2
2     3
dtype: int64

In [133]: se[5] = 5.

In [134]: se
Out[134]:
0     1.0
1     2.0
2     3.0
5     5.0
dtype: float64
```

A DataFrame can be enlarged on either axis via `.loc`.

```
In [135]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
                      columns=['A', 'B'])

In [136]: dfi
Out[136]:
   A  B
0  0  1
1  2  3
```

(continues on next page)
In [137]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [138]: dfi
Out[138]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4

This is like an append operation on the DataFrame.

In [139]: dfi.loc[3] = 5

In [140]: dfi
Out[140]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5

### 2.5.12 Fast scalar value getting and setting

Since indexing with [] must handle a lot of cases (single-label access, slicing, boolean indexing, etc.), it has a bit of overhead in order to figure out what you’re asking for. If you only want to access a scalar value, the fastest way is to use the at and iat methods, which are implemented on all of the data structures.

Similarly to loc, at provides label based scalar lookups, while, iat provides integer based lookups analogously to iloc.

In [141]: s.iat[5]
Out[141]: 5

In [142]: df.at[dates[5], 'A']
Out[142]: -0.6736897080883706

In [143]: df.iat[3, 0]
Out[143]: 0.7215551622443669

You can also set using these same indexers.

In [144]: df.at[dates[5], 'E'] = 7

In [145]: df.iat[3, 0] = 7

at may enlarge the object in-place as above if the indexer is missing.

In [146]: df.at[dates[-1] + pd.Timedelta('1 day'), 0] = 7

In [147]: df
2.5.13 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: `|` for `or`, `&` for `and`, and `~` for `not`. These must be grouped by using parentheses, since by default Python will evaluate an expression such as `df['A'] > 2 & df['B'] < 3` as `df['A'] > (2 & df['B']) < 3`, while the desired evaluation order is `(df['A'] > 2) & (df['B'] < 3)`.

Using a boolean vector to index a Series works exactly as in a NumPy ndarray:

```python
In [148]: s = pd.Series(range(-3, 4))
In [149]: s
Out[149]:
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int64
In [150]: s[s > 0]
Out[150]:
4    1
5    2
6    3
dtype: int64
In [151]: s[(s < -1) | (s > 0.5)]
Out[151]:
0   -3
1   -2
4    1
5    2
6    3
dtype: int64
In [152]: s[~(s < 0)]
```

(continues on next page)
You may select rows from a DataFrame using a boolean vector the same length as the DataFrame’s index (for example, something derived from one of the columns of the DataFrame):

```
In [153]: df[df['A'] > 0]
```

```
Out[153]:
          A       B       C      D       E   
2000-01-01 0.469112 -0.282863 -1.509059 -1.135632 NaN
2000-01-02 1.212112 -0.173215  0.119209 -1.044236 NaN
2000-01-04 7.000000 -0.706771 -1.039575  0.271860 NaN
2000-01-07 0.404705  0.577046 -1.715002 -1.039268 NaN
```

List comprehensions and the map method of Series can also be used to produce more complex criteria:

```
In [154]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                        'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                        'c': np.random.randn(7)})

# only want 'two' or 'three'
In [155]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [156]: df2[criterion]
```

```
   a   b   c
2  two y 0.041290
3  three x 0.361719
4  two y -0.238075
```

# equivalent but slower
```
In [157]: df2[[x.startswith('t') for x in df2['a']]]
```

```
   a   b   c
2  two y 0.041290
3  three x 0.361719
4  two y -0.238075
```

# Multiple criteria
```
In [158]: df2[criterion & (df2['b'] == 'x')]
```

```
   a   b   c
3  three x 0.361719
```

With the choice methods Selection by Label, Selection by Position, and Advanced Indexing you may select along more than one axis using boolean vectors combined with other indexing expressions.
In [159]: df2.loc[criterion & (df2['b'] == 'x'), 'b': 'c']
Out[159]:
   b   c
3  x  0.361719

**Warning:** `iloc` supports two kinds of boolean indexing. If the indexer is a boolean `Series`, an error will be raised. For instance, in the following example, `df.iloc[s.values, 1]` is ok. The boolean indexer is an array. But `df.iloc[s, 1]` would raise `ValueError`.

In [160]: df = pd.DataFrame([[1, 2], [3, 4], [5, 6]],
......:     index=list('abc'),
......:     columns=['A', 'B'])

In [161]: s = (df['A'] > 2)

In [162]: s
Out[162]:
a  False
b  True
c  True
Name: A, dtype: bool

In [163]: df.loc[s, 'B']
Out[163]:
   b  c
b  4  6
c  6
Name: B, dtype: int64

In [164]: df.iloc[s.values, 1]
Out[164]:
   b  c
b  4  6
c  6
Name: B, dtype: int64

2.5.14 Indexing with `isin`

Consider the `isin()` method of `Series`, which returns a boolean vector that is true wherever the `Series` elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

In [165]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')

In [166]: s
Out[166]:
   0  1  2  3  4
4  0
3  1
2  2
1  3
0  4
dtype: int64

(continues on next page)
In [167]: s.isin([2, 4, 6])
Out[167]:
4   False
3   False
2    True
1   False
0    True
dtype: bool

In [168]: s[s.isin([2, 4, 6])]
Out[168]:
   2
0  4
dtype: int64

The same method is available for Index objects and is useful for the cases when you don’t know which of the sought labels are in fact present:

In [169]: s[s.index.isin([2, 4, 6])]
Out[169]:
   4
0  0
2  2
dtype: int64

# compare it to the following
In [170]: s.reindex([2, 4, 6])
Out[170]:
   2 2.0
   4 0.0
   6 NaN
dtype: float64

In addition to that, MultiIndex allows selecting a separate level to use in the membership check:

In [171]: s_mi = pd.Series(np.arange(6),
              index=pd.MultiIndex.from_product([['a', 'b', 'c'], [0, 1]])

In [172]: s_mi
Out[172]:
   0  a  0
   b  1
   c  2
1  a  3
   b  4
   c  5
dtype: int64

In [173]: s_mi.loc[s_mi.index.isin([[1, 'a'], (2, 'b'), (0, 'c')]])
Out[173]:
   0  c  2

(continues on next page)
DataFrame also has an `isin()` method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

```python
In [175]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
                      'ids2': ['a', 'n', 'c', 'n']})

In [176]: values = ['a', 'b', 1, 3]

In [177]: df.isin(values)
Out[177]:
   vals  ids  ids2
0   True  True  True
1   False  True  False
2   True  False  False
3   False  False  False
```

Oftentimes you'll want to match certain values with certain columns. Just make values a dict where the key is the column, and the value is a list of items you want to check for.

```python
In [178]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [179]: df.isin(values)
Out[179]:
   vals  ids  ids2
0   True  True  True
1   False  True  False
2   True  False  False
3   False  False  False
```

To return the DataFrame of booleans where the values are not in the original DataFrame, use the ~ operator:

```python
In [180]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [181]: ~df.isin(values)
Out[181]:
   vals  ids  ids2
0   False  False  True
1   True  False  True
2   False  True  True
3   True  True  True
```
Combine DataFrame's `isin` with the `any()` and `all()` methods to quickly select subsets of your data that meet a given criteria. To select a row where each column meets its own criterion:

```python
In [182]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}
In [183]: row_mask = df.isin(values).all(1)
In [184]: df[row_mask]
Out[184]:
   vals  ids  ids2
0     1    a     a
```

### 2.5.15 The `where()` Method and Masking

Selecting values from a Series with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows:

```python
In [185]: s[s > 0]
Out[185]:
3    1
2    2
1    3
0    4
dtype: int64
```

To return a Series of the same shape as the original:

```python
In [186]: s.where(s > 0)
Out[186]:
4  NaN
3  1.0
2  2.0
1  3.0
0  4.0
dtype: float64
```

Selecting values from a DataFrame with a boolean criterion now also preserves input data shape. `where` is used under the hood as the implementation. The code below is equivalent to `df.where(df < 0)`.

```python
In [187]: df[df < 0]
Out[187]:
   A         B         C         D
2000-01-01 -2.104139 -1.309525 NaN       NaN
2000-01-02 -0.352480 NaN -1.192319       NaN
2000-01-03 -0.864883 NaN -0.227870       NaN
2000-01-04 NaN -1.222082 NaN       NaN
2000-01-05 NaN -0.065656 -1.169184       NaN
2000-01-06 NaN -0.948458 NaN -0.684718
2000-01-07 -2.670153 -0.114722 NaN -0.048048
2000-01-08 NaN NaN -0.048788 -0.808838
```
In addition, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```python
In [188]: df.where(df < 0, -df)
Out[188]:
   A         B         C         D
2000-01-01 -2.104139 -1.309525 -0.485855 -0.245166
2000-01-02 -0.352480 -0.390389 -1.192319 -1.655824
2000-01-03 -0.864883 -0.299674 -0.227870 -0.281059
2000-01-04 -0.846958 -1.222082 -0.600705 -1.233203
2000-01-05 -0.669692 -0.605656 -1.169184 -0.342416
2000-01-06 -0.868584 -0.948458 -2.297780 -0.684718
2000-01-07 -2.670153 -0.114722 -0.168904 -0.048048
2000-01-08 -0.801196 -1.392071 -0.048788 -0.808838
```

You may wish to set values based on some boolean criteria. This can be done intuitively like so:

```python
In [189]: s2 = s.copy()
In [190]: s2[s2 < 0] = 0
In [191]: s2
Out[191]:
4  0
3  1
2  2
1  3
0  4
dtype: int64
In [192]: df2 = df.copy()
In [193]: df2[df2 < 0] = 0
In [194]: df2
Out[194]:
   A         B         C         D
2000-01-01 0.000000 0.000000 0.485855 0.245166
2000-01-02 0.000000 0.390389 0.000000 1.655824
2000-01-03 0.000000 0.299674 0.000000 0.281059
2000-01-04 0.846958 0.000000 0.600705 0.000000
2000-01-05 0.669692 0.000000 0.000000 0.342416
2000-01-06 0.868584 0.000000 2.297780 0.000000
2000-01-07 0.000000 0.000000 0.168904 0.000000
2000-01-08 0.801196 1.392071 0.000000 0.000000
```

By default, `where` returns a modified copy of the data. There is an optional parameter `inplace` so that the original data can be modified without creating a copy:

```python
In [195]: df_orig = df.copy()
In [196]: df_orig.where(df > 0, -df, inplace=True)
In [197]: df_orig
```

(continues on next page)
Note: The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2).

In [198]: df.where(df < 0, -df) == np.where(df < 0, df, -df)

Out[198]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Alignment

Furthermore, `where` aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via `.loc` (but on the contents rather than the axis labels).

In [199]: df2 = df.copy()

In [200]: df2[df2[1:4] > 0] = 3

In [201]: df2

Out[201]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-2.104139</td>
<td>-1.309525</td>
<td>0.485855</td>
<td>0.245166</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.352480</td>
<td>0.390389</td>
<td>1.192319</td>
<td>1.655824</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.864883</td>
<td>0.299674</td>
<td>0.227870</td>
<td>0.281059</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.846958</td>
<td>1.222082</td>
<td>0.600705</td>
<td>1.233203</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.669692</td>
<td>0.605656</td>
<td>1.169184</td>
<td>0.342416</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.868584</td>
<td>0.948458</td>
<td>2.297780</td>
<td>0.684718</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>2.670153</td>
<td>0.114722</td>
<td>0.168904</td>
<td>0.048048</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.801196</td>
<td>1.392071</td>
<td>0.048788</td>
<td>0.808838</td>
</tr>
</tbody>
</table>

Where can also accept `axis` and `level` parameters to align the input when performing the where.

In [202]: df2 = df.copy()
In [203]: df2.where(df2 > 0, df2['A'], axis='index')

Out[203]:
          A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02  0.390389  0.390389  0.352480  1.655824
2000-01-03 -0.864883 -0.864883  0.299674  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196

This is equivalent to (but faster than) the following.

In [204]: df2 = df.copy()

In [205]: df.apply(lambda x, y: x.where(x > 0, y), y=df['A'])

Out[205]:
          A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02  0.390389  0.390389  0.352480  1.655824
2000-01-03 -0.864883 -0.864883  0.299674  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196

where can accept a callable as condition and other arguments. The function must be with one argument (the calling Series or DataFrame) and that returns valid output as condition and other argument.

In [206]: df3 = pd.DataFrame({'A': [1, 2, 3],
                        'B': [4, 5, 6],
                        'C': [7, 8, 9]})

In [207]: df3.where(lambda x: x > 4, lambda x: x + 10)

Out[207]:
          A         B         C
0    11  14  14
1    12  5  12
2    13  6  16
Mask

`mask()` is the inverse boolean operation of `where`.

```python
In [208]: s.mask(s >= 0)
Out[208]:
4  NaN
3  NaN
2  NaN
1  NaN
0  NaN
dtype: float64
```

```python
In [209]: df.mask(df >= 0)
Out[209]:
     A         B         C         D
2000-01-01 -2.184139 -1.309525 NaN        NaN
2000-01-02 -0.352480 NaN  -1.192319        NaN
2000-01-03  0.864883  NaN   -0.227870        NaN
2000-01-04  NaN     -1.222082 NaN   -1.233203
2000-01-05  NaN     -0.605656 NaN  -1.169184
2000-01-06  NaN     -0.948458  NaN     -0.684718
2000-01-07  NaN     -2.670153  NaN     -0.048788
2000-01-08  NaN     0.048788  NaN     -0.808838
```

### 2.5.16 Setting with enlargement conditionally using `numpy()`

An alternative to `where()` is to use `numpy.where()`. Combined with setting a new column, you can use it to enlarge a DataFrame where the values are determined conditionally.

Consider you have two choices to choose from in the following DataFrame. And you want to set a new column color to ‘green’ when the second column has ‘Z’. You can do the following:

```python
In [210]: df = pd.DataFrame({'col1': list('ABBC'), 'col2': list('ZZXY'))

In [211]: df['color'] = np.where(df['col2'] == 'Z', 'green', 'red')

In [212]: df
Out[212]:
     col1 col2  color
0      A    Z   green
1      B    Z   green
2      B    X    red
3      C    Y    red
```

If you have multiple conditions, you can use `numpy.select()` to achieve that. Say corresponding to three conditions there are three choice of colors, with a fourth color as a fallback, you can do the following.

```python
In [213]: conditions = [
            ....:     (df['col2'] == 'Z') & (df['col1'] == 'A'),
            ....:     (df['col2'] == 'Z') & (df['col1'] == 'B'),
            ....:     (df['col1'] == 'B')
            ....: ]
```

(continues on next page)
.....:

```
In [214]: choices = ['yellow', 'blue', 'purple']

In [215]: df['color'] = np.select(conditions, choices, default='black')

In [216]: df
```

```
Out[216]:
   col1 col2  color
0     A   Z  yellow
1     B   Z   blue
2     B   X  purple
3     C   Y  black
```

### 2.5.17 The `query()` Method

`DataFrame` objects have a `query()` method that allows selection using an expression.

You can get the value of the frame where column `b` has values between the values of columns `a` and `c`. For example:

```
In [217]: n = 10

In [218]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [219]: df
```

```
Out[219]:
     a       b       c
0  0.438921  0.118680  0.863670
1  0.138138  0.577363  0.686602
2  0.595307  0.564592  0.520630
3  0.913052  0.926075  0.616184
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
6  0.792342  0.216974  0.564056
7  0.397890  0.454131  0.915716
8  0.074315  0.437913  0.019794
9  0.559209  0.502065  0.026437
```

# pure python
```
In [220]: df[(df['a'] < df['b']) & (df['b'] < df['c'])]
```

```
Out[220]:
     a       b       c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716
```

# query
```
In [221]: df.query('(a < b) & (b < c)')
```

```
Out[221]:
     a       b       c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716
```
Do the same thing but fall back on a named index if there is no column with the name \( a \).

```
In [222]: df = pd.DataFrame(np.random.randint(n / 2, size=(n, 2)), columns=list('bc'))
In [223]: df.index.name = 'a'
In [224]: df
Out[224]:
   b  c
0  0  4
1  0  1
2  3  4
3  4  3
4  1  4
5  0  3
6  0  1
7  3  4
8  2  3
9  1  1
```

If instead you don’t want to or cannot name your index, you can use the name `index` in your query expression:

```
In [225]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)), columns=list('bc'))
In [226]: df
Out[226]:
   b  c
0  3  1
1  3  0
2  5  6
3  5  2
4  7  4
5  0  1
6  2  5
7  0  1
8  6  0
9  7  9
```

```
In [227]: df.query('index < b < c')
Out[227]:
   b  c
2  3  4
```

(continues on next page)
Note: If the name of your index overlaps with a column name, the column name is given precedence. For example,

```python
In [229]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})
In [230]: df.index.name = 'a'
In [231]: df.query('a > 2')  # uses the column 'a', not the index
Out[231]:
   a
0  3
1  1
2  3
3  3
```

You can still use the index in a query expression by using the special identifier 'index':

```python
In [232]: df.query('index > 2')
Out[232]:
   a
0  3
1  2
```

If for some reason you have a column named index, then you can refer to the index as ilevel_0 as well, but at this point you should consider renaming your columns to something less ambiguous.

**MultiIndex query() Syntax**

You can also use the levels of a DataFrame with a MultiIndex as if they were columns in the frame:

```python
In [233]: n = 10
In [234]: colors = np.random.choice(['red', 'green'], size=n)
In [235]: foods = np.random.choice(['eggs', 'ham'], size=n)
In [236]: colors
Out[236]:
array(['red', 'red', 'red', 'green', 'green', 'green', 'green', 'green', 'green', 'green'], dtype='<U5')
In [237]: foods
Out[237]:
array(['ham', 'ham', 'eggs', 'eggs', 'eggs', 'eggs', 'ham', 'ham', 'eggs', 'eggs',
       'eggs'], dtype='<U4')
In [238]: index = pd.MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
```
In [239]: df = pd.DataFrame(np.random.randn(n, 2), index=index)

In [240]: df
Out[240]:
   0     1
color food
red  ham 0.194889 -0.381994
    ham  0.318587  2.089075
    eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
    eggs -2.029766  0.792652
    ham  0.461007 -0.542749
    ham -0.305384 -0.479195
    eggs  0.095031 -0.270099
    eggs -0.707140 -0.773882
    eggs  0.229453  0.304418

In [241]: df.query('color == "red"')
Out[241]:
   0     1
color food
red  ham 0.194889 -0.381994
    ham  0.318587  2.089075
    eggs -0.728293 -0.090255

If the levels of the MultiIndex are unnamed, you can refer to them using special names:

In [242]: df.index.names = [None, None]

In [243]: df
Out[243]:
   0     1
red  ham 0.194889 -0.381994
    ham  0.318587  2.089075
    eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
    eggs -2.029766  0.792652
    ham  0.461007 -0.542749
    ham -0.305384 -0.479195
    eggs  0.095031 -0.270099
    eggs -0.707140 -0.773882
    eggs  0.229453  0.304418

In [244]: df.query('ilevel_0 == "red"')
Out[244]:
   0     1
red  ham 0.194889 -0.381994
    ham  0.318587  2.089075
    eggs -0.728293 -0.090255

The convention is ilevel_0, which means “index level 0” for the 0th level of the index.
query() Use Cases

A use case for `query()` is when you have a collection of `DataFrame` objects that have a subset of column names (or index levels/names) in common. You can pass the same query to both frames without having to specify which frame you're interested in querying.

```python
In [245]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [246]: df
Out[246]:
   a     b     c
0  0.224283  0.736107  0.139168
1  0.302827  0.657803  0.713897
2  0.611185  0.136624  0.984960
3  0.195246  0.123436  0.627712
4  0.618673  0.371660  0.047902
5  0.480088  0.062993  0.185760
6  0.568018  0.483467  0.445289
7  0.309040  0.274580  0.587101
8  0.258993  0.477769  0.370255
9  0.550459  0.840870  0.304611

In [247]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)

In [248]: df2
Out[248]:
   a     b     c
0  0.357579  0.229800  0.596001
1  0.309059  0.957923  0.965663
2  0.123102  0.336914  0.318616
3  0.526506  0.323321  0.860813
4  0.518736  0.486514  0.384724
5  0.190804  0.505723  0.614533
6  0.891939  0.623977  0.676639
7  0.480559  0.378528  0.460858
8  0.420223  0.136404  0.141295
9  0.732206  0.419540  0.604675
10 0.604466  0.848974  0.896165
11 0.589168  0.920046  0.732716

In [249]: expr = '0.0 <= a <= c <= 0.5'

In [250]: map(lambda frame: frame.query(expr), [df, df2])
Out[250]: <map at 0x7f2d1f5c8160>
```
query() Python versus pandas Syntax Comparison

Full numpy-like syntax:

```python
In [251]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=list('abc'))

In [252]: df
Out[252]:
   a  b  c
0  7  8  9
1  1  0  7
2  2  7  2
3  6  2  2
4  2  6  3
5  3  8  2
6  1  7  2
7  5  1  5
8  9  8  0
9  1  5  0

In [253]: df.query('(a < b) & (b < c)')
Out[253]:
   a  b  c
0  7  8  9

In [254]: df[(df['a'] < df['b']) & (df['b'] < df['c'])]
Out[254]:
   a  b  c
0  7  8  9
```

Slightly nicer by removing the parentheses (comparison operators bind tighter than & and |):

```python
In [255]: df.query('a < b & b < c')
Out[255]:
   a  b  c
0  7  8  9
```

Use English instead of symbols:

```python
In [256]: df.query('a < b and b < c')
Out[256]:
   a  b  c
0  7  8  9
```

Pretty close to how you might write it on paper:

```python
In [257]: df.query('a < b < c')
Out[257]:
   a  b  c
0  7  8  9
```
The **in** and **not in** operators

`query()` also supports special use of Python’s **in** and **not in** comparison operators, providing a succinct syntax for calling the `isin` method of a `Series` or `DataFrame`.

```python
# get all rows where columns "a" and "b" have overlapping values
In [258]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbbcccc'),
                        'c': np.random.randint(5, size=12),
                        'd': np.random.randint(9, size=12))

In [259]: df
Out[259]:
   a    b    c    d
0  a    a  2    6
1  a    a  4    7
2  b    a  1    6
3  b    a  2    1
4  c    b  3    6
5  c    b  0    2
6  d    b  3    3
7  d    b  2    1
8  e    c  4    3
9  e    c  2    0
10 f    c  0    6
11 f    c  1    2

In [260]: df.query('a in b')
Out[260]:
   a    b    c    d
0  a    a  2    6
1  a    a  4    7
2  b    a  1    6
3  b    a  2    1
4  c    b  3    6
5  c    b  0    2

# How you'd do it in pure Python
In [261]: df[df['a'].isin(df['b'])]
Out[261]:
   a    b    c    d
0  a    a  2    6
1  a    a  4    7
2  b    a  1    6
3  b    a  2    1
4  c    b  3    6
5  c    b  0    2

In [262]: df.query('a not in b')
Out[262]:
   a    b    c    d
6  d    b  3    3
7  d    b  2    1

(continues on next page)
```
# pure Python

```python
In [263]: df[~df['a'].isin(df['b'])]
```

Out[263]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>e</td>
<td>c</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>c</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>c</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

You can combine this with other expressions for very succinct queries:

```python
# rows where cols a and b have overlapping values  
# and col c's values are less than col d's
In [264]: df.query('a in b and c < d')
```

Out[264]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>b</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>b</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

```python
# pure Python
In [265]: df[df['b'].isin(df['a']) & (df['c'] < df['d'])]
```

Out[265]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>b</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>b</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Note: Note that `in` and `not in` are evaluated in Python, since `numexpr` has no equivalent of this operation. However, only the `in`/`not in` expression itself is evaluated in vanilla Python. For example, in the expression

```python
df.query('a in b + c + d')
```

(b + c + d) is evaluated by `numexpr` and then the `in` operation is evaluated in plain Python. In general, any operations that can be evaluated using `numexpr` will be.
Special use of the == operator with list objects

Comparing a list of values to a column using ==/!= works similarly to in/not in.

```
In [266]: df.query('b == ["a", "b", "c"]')
Out[266]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2
6  d  b  3  3
7  d  b  2  1
8  e  c  4  3
9  e  c  2  0
10 f  c  0  6
11 f  c  1  2

# pure Python
In [267]: df[df['b'].isin(["a", "b", "c")]]
Out[267]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2
6  d  b  3  3
7  d  b  2  1
8  e  c  4  3
9  e  c  2  0
10 f  c  0  6
11 f  c  1  2

In [268]: df.query('c == [1, 2]')
Out[268]:
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2

In [269]: df.query('c != [1, 2]')
Out[269]:
   a  b  c  d
1  a  a  4  7
4  c  b  3  6
5  c  b  0  2
```

(continues on next page)
# using in/not in

```python
In [270]: df.query('1, 2 in c')
Out[270]:
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2
```

```python
In [271]: df.query('1, 2 not in c')
Out[271]:
   a  b  c  d
1  a  a  4  7
4  c  b  3  6
5  c  b  0  2
6  d  b  3  3
8  e  c  4  3
10 f  c  0  6
```

# pure Python

```python
In [272]: df[df['c'].isin([1, 2])]
Out[272]:
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2
```

### Boolean operators

You can negate boolean expressions with the word `not` or the `~` operator.

```python
In [273]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))
In [274]: df['bools'] = np.random.rand(len(df)) > 0.5
In [275]: df.query('~bools')
Out[275]:
   a  b  c  bools
2 0.697753 0.212799 0.329209 False
7 0.275396 0.691034 0.826619 False
8 0.190649 0.558748 0.262467 False
```

(continues on next page)
In [276]: df.query('not bools')
Out[276]:
        a     b     c  bools
2  0.697753  0.212799  0.329209  False
7  0.275396  0.691034  0.826619  False
8  0.190649  0.558748  0.262467  False

In [277]: df.query('not bools') == df[~df['bools']]
Out[277]:
        a    b    c  bools
2  True  True  True  True
7  True  True  True  True
8  True  True  True  True

Of course, expressions can be arbitrarily complex too:

# short query syntax
In [278]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [279]: longer = df[(df['a'] < df['b']) & (df['b'] < df['c']) & (~df['bools']) | (df['bools'] > 2)]

In [280]: shorter
Out[280]:
        a     b     c  bools
7  0.275396  0.691034  0.826619  False

In [281]: longer
Out[281]:
        a     b     c  bools
7  0.275396  0.691034  0.826619  False
Performance of `query()`

`DataFrame.query()` using `numexpr` is slightly faster than Python for large frames.

**Note:** You will only see the performance benefits of using the `numexpr` engine with `DataFrame.query()` if your frame has more than approximately 200,000 rows.

This plot was created using a `DataFrame` with 3 columns each containing floating point values generated using `numpy.random.randn()`.
### 2.5.18 Duplicate data

If you want to identify and remove duplicate rows in a DataFrame, there are two methods that will help: `duplicated` and `drop_duplicates`. Each takes as an argument the columns to use to identify duplicated rows.

- `duplicated` returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.
- `drop_duplicates` removes duplicate rows.

By default, the first observed row of a duplicate set is considered unique, but each method has a `keep` parameter to specify targets to be kept.

- `keep='first'` (default): mark / drop duplicates except for the first occurrence.
- `keep='last'`: mark / drop duplicates except for the last occurrence.
- `keep=False`: mark / drop all duplicates.

```python
In [283]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two', 'three', 'four'],
                        'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
                        'c': np.random.randn(7)})

In [284]: df2
Out[284]:
      a  b    c
0     one  x -1.067137
1     one  y   0.309500
2     two  x  -0.211056
3     two  y  -1.842023
4     two  x  -0.390820
5    three  x -1.964475
6      four  x   1.298329

In [285]: df2.duplicated('a')
Out[285]:
0   False
1    True
2   False
3    True
4    True
5   False
6   False
dtype: bool

In [286]: df2.duplicated('a', keep='last')
Out[286]:
0   True
1    False
2    True
3    True
4   False
5   False
6   False
dtype: bool
```

(continues on next page)
In [287]: df2.duplicated('a', keep=False)
Out[287]:
          0   1   2   3   4   5   6
dtype: bool
   0   True  True  True  True  True  False
   1   False

In [288]: df2.drop_duplicates('a')
Out[288]:
   a   b   c
  0 one   x -1.067137
  2 two   x -0.211056
  5 three  x -1.964475
  6 four  x  1.298329

In [289]: df2.drop_duplicates('a', keep='last')
Out[289]:
   a   b   c
  1 one   y  0.309500
  4 two   x -0.390820
  5 three  x -1.964475
  6 four  x  1.298329

Also, you can pass a list of columns to identify duplications.

In [291]: df2.duplicated(['a', 'b'])
Out[291]:
          0   1   2   3
 dtype: bool
   0   False  False  False
   1   False
   2   False
   3   False
   4   True
   5   False
   6   False

In [292]: df2.drop_duplicates(['a', 'b'])
Out[292]:
   a   b   c
  0 one   x -1.067137
  1 one   y  0.309500
To drop duplicates by index value, use `Index.duplicated` then perform slicing. The same set of options are available for the `keep` parameter.

```
In [293]: df3 = pd.DataFrame({'a': np.arange(6),
                           'b': np.random.randn(6),
                           index=['a', 'a', 'b', 'c', 'b', 'a'])

In [294]: df3
Out[294]:
    a   b
   a  0  1.440455
   a  1  2.456086
   b  2  1.038402
   c  3  0.894409
   b  4  0.683536
   a  5  3.082764

In [295]: df3.index.duplicated()
Out[295]: array([False,  True,  False,  False,  True,  True])

In [296]: df3[~df3.index.duplicated()]
Out[296]:
    a   b
   a  0  1.440455
   b  2  1.038402
   c  3  0.894409

In [297]: df3[~df3.index.duplicated(keep='last')]
Out[297]:
    a   b
   c  3  0.894409
   b  4  0.683536
   a  5  3.082764

In [298]: df3[~df3.index.duplicated(keep=False)]
Out[298]:
    a   b
   c  3  0.894409
```
2.5.19 Dictionary-like get() method

Each of Series or DataFrame have a get method which can return a default value.

```python
In [299]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [300]: s.get('a')  # equivalent to s['a']
Out[300]: 1

In [301]: s.get('x', default=-1)
Out[301]: -1
```

2.5.20 Looking up values by index/column labels

Sometimes you want to extract a set of values given a sequence of row labels and column labels, this can be achieved by pandas.factorize and NumPy indexing. For instance:

```python
In [302]: df = pd.DataFrame({'col': ['A', 'A', 'B', 'B'],
                         'A': [80, 23, np.nan, 22],
                         'B': [80, 55, 76, 67]})

In [303]: df
Out[303]:
   col  A    B
0    A  80.0  80
1    A  23.0  55
2    B  NaN  76
3    B  22.0  67

In [304]: idx, cols = pd.factorize(df['col'])

In [305]: df.reindex(cols, axis=1).to_numpy()[np.arange(len(df)), idx]
Out[305]: array([80., 23., 76., 67.])
```

Formerly this could be achieved with the dedicated DataFrame.lookup method which was deprecated in version 1.2.0.

2.5.21 Index objects

The pandas Index class and its subclasses can be viewed as implementing an ordered multiset. Duplicates are allowed. However, if you try to convert an Index object with duplicate entries into a set, an exception will be raised.

Index also provides the infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create an Index directly is to pass a list or other sequence to Index:

```python
In [306]: index = pd.Index(['e', 'd', 'a', 'b'])

In [307]: index
Out[307]: Index(['e', 'd', 'a', 'b'], dtype='object')

In [308]: 'd' in index
Out[308]: True
```
You can also pass a name to be stored in the index:

```python
In [309]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')
In [310]: index.name
Out[310]: 'something'
```

The name, if set, will be shown in the console display:

```python
In [311]: index = pd.Index(list(range(5)), name='rows')
In [312]: columns = pd.Index(['A', 'B', 'C'], name='cols')
In [313]: df = pd.DataFrame(np.random.randn(5, 3), index=index, columns=columns)
In [314]: df
Out[314]:
   A     B     C
rows
0  1.295989 -1.051694 1.340429
1 -2.366110  0.428241  0.387275
2  0.433306  0.929548  0.278094
3  2.154730 -0.315628  0.264223
4  1.126818  1.132290 -0.353310
```

### Setting metadata

Indexes are “mostly immutable”, but it is possible to set and change their name attribute. You can use the `rename`, `set_names` to set these attributes directly, and they default to returning a copy.

See Advanced Indexing for usage of MultiIndexes.

```python
In [316]: ind = pd.Index([1, 2, 3])
In [317]: ind.rename("apple")
Out[317]: Int64Index([1, 2, 3], dtype='int64', name='apple')
In [318]: ind
Out[318]: Int64Index([1, 2, 3], dtype='int64')
In [319]: ind.set_names(["apple"], inplace=True)
In [320]: ind.name = "bob"
```
In [321]: ind
Out[321]: Int64Index([1, 2, 3], dtype='int64', name='bob')

set_names, set_levels, and set_codes also take an optional level argument

In [322]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first', 'second'])

In [323]: index
Out[323]: MultiIndex([(0, 'one'), (0, 'two'), (1, 'one'), (1, 'two'), (2, 'one'), (2, 'two')], names=['first', 'second'])

In [324]: index.levels[1]
Out[324]: Index(['one', 'two'], dtype='object', name='second')

In [325]: index.set_levels(['a', 'b'], level=1)
Out[325]: MultiIndex([(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')], names=['first', 'second'])

Set operations on Index objects

The two main operations are union and intersection. Difference is provided via the .difference() method.

In [326]: a = pd.Index(['c', 'b', 'a'])

In [327]: b = pd.Index(['c', 'e', 'd'])

In [328]: a.difference(b)
Out[328]: Index(['a', 'b'], dtype='object')

Also available is the symmetric_difference operation, which returns elements that appear in either idx1 or idx2, but not in both. This is equivalent to the Index created by idx1.difference(idx2).union(idx2.difference(idx1)), with duplicates dropped.

In [329]: idx1 = pd.Index([1, 2, 3, 4])

In [330]: idx2 = pd.Index([2, 3, 4, 5])
In [331]: idx1.symmetric_difference(idx2)
Out[331]: Int64Index([1, 5], dtype='int64')

Note: The resulting index from a set operation will be sorted in ascending order.

When performing `Index.union()` between indexes with different dtypes, the indexes must be cast to a common dtype. Typically, though not always, this is object dtype. The exception is when performing a union between integer and float data. In this case, the integer values are converted to float.

In [332]: idx1 = pd.Index([0, 1, 2])
In [333]: idx2 = pd.Index([0.5, 1.5])
In [334]: idx1.union(idx2)
Out[334]: Float64Index([0.0, 0.5, 1.0, 1.5, 2.0], dtype='float64')

Missing values

Important: Even though `Index` can hold missing values (NaN), it should be avoided if you do not want any unexpected results. For example, some operations exclude missing values implicitly.

`Index.fillna` fills missing values with specified scalar value.

In [335]: idx1 = pd.Index([1, np.nan, 3, 4])
In [336]: idx1
Out[336]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')
In [337]: idx1.fillna(2)
Out[337]: Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')

In [339]: idx2
Out[339]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'], dtype='datetime64[ns]', freq=None)
In [340]: idx2.fillna(pd.Timestamp('2011-01-02'))
Out[340]: DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], dtype='datetime64[ns]', freq=None)
2.5.22 Set / reset index

Occasionally you will load or create a data set into a DataFrame and want to add an index after you’ve already done so. There are a couple of different ways.

Set an index

Dataframe has a set_index() method which takes a column name (for a regular Index) or a list of column names (for a MultiIndex). To create a new, re-indexed DataFrame:

```python
In [341]: data
Out[341]:
     a    b    c    d
0   bar  one  z  1.0
1   bar  two  y  2.0
2   foo  one  x  3.0
3   foo  two  w  4.0

In [342]: indexed1 = data.set_index('c')

In [343]: indexed1
Out[343]:
     a    b   d
    c
   z  bar  one  1.0
   y  bar  two  2.0
   x  foo  one  3.0
   w  foo  two  4.0

In [344]: indexed2 = data.set_index(['a', 'b'])

In [345]: indexed2
Out[345]:
     c   d
    a  b
   bar  one  z  1.0
       two  y  2.0
   foo  one  x  3.0
       two  w  4.0
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```python
In [346]: frame = data.set_index('c', drop=False)

In [347]: frame = frame.set_index(['a', 'b'], append=True)

In [348]: frame
Out[348]:
     c   d
    a  b
   bar  one  z  1.0
       two  y  2.0
   foo  one  x  3.0
       two  w  4.0
```

(continues on next page)
Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):

```python
In [349]: data.set_index('c', drop=False)
Out[349]:
   a   b   c    d
 c
z bar one z 1.0
y bar two y 2.0
x foo one x 3.0
w foo two w 4.0
```

```python
In [350]: data.set_index(['a', 'b'], inplace=True)
```

```python
In [351]: data
Out[351]:
   c    d
   a   b
bar one z 1.0
   two y 2.0
foo one x 3.0
   two w 4.0
```

## Reset the index

As a convenience, there is a new function on DataFrame called `reset_index()` which transfers the index values into the DataFrame’s columns and sets a simple integer index. This is the inverse operation of `set_index()`.

```python
In [352]: data
Out[352]:
   c    d
   a   b
bar one z 1.0
   two y 2.0
foo one x 3.0
   two w 4.0
```

```python
In [353]: data.reset_index()
Out[353]:
   a   b   c    d
 0 bar one z 1.0
 1 bar two y 2.0
 2 foo one x 3.0
 3 foo two w 4.0
```

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the `names` attribute.

You can use the `level` keyword to remove only a portion of the index:
In [354]: frame
Out[354]:
   c  d
   c  a  b
   z  bar  one  z  1.0
   y  bar  two  y  2.0
   x  foo  one  x  3.0
   w  foo  two  w  4.0

In [355]: frame.reset_index(level=1)
Out[355]:
   a  c  d
   c  b
   z  one  bar  z  1.0
   y  two  bar  y  2.0
   x  one  foo  x  3.0
   w  two  foo  w  4.0

reset_index takes an optional parameter drop which if true simply discards the index, instead of putting index values in the DataFrame’s columns.

Adding an ad hoc index

If you create an index yourself, you can just assign it to the index field:

data.index = index

2.5.23 Returning a view versus a copy

When setting values in a pandas object, care must be taken to avoid what is called chained indexing. Here is an example.

In [356]: dfmi = pd.DataFrame([list('abcd'),
       ....: list('efgh'),
       ....: list('ijkl'),
       ....: list('mnop')],
       ....: columns=pd.MultiIndex.from_product([[one, two],
       ....:                                       [first, second]]))

In [357]: dfmi
Out[357]:
   one  two
   first  second  first  second
   0     a  b     c  d
   1     e  f     g  h
   2     i  j     k  l
   3     m  n     o  p

Compare these two access methods:
In [358]: dfmi['one']['second']
Out[358]:
0 b
1 f
2 j
3 n
Name: second, dtype: object

In [359]: dfmi.loc[:, ('one', 'second')]
Out[359]:
0 b
1 f
2 j
3 n
Name: (one, second), dtype: object

These both yield the same results, so which should you use? It is instructive to understand the order of operations on these and why method 2 (.loc) is much preferred over method 1 (chained []).

dfmi['one'] selects the first level of the columns and returns a DataFrame that is singly-indexed. Then another Python operation dfmi_with_one['second'] selects the series indexed by 'second'. This is indicated by the variable dfmi_with_one because pandas sees these operations as separate events. e.g. separate calls to __getitem__, so it has to treat them as linear operations, they happen one after another.

Contrast this to df.loc[:,('one','second')] which passes a nested tuple of (slice(None),('one', 'second')) to a single call to __getitem__. This allows pandas to deal with this as a single entity. Furthermore this order of operations can be significantly faster, and allows one to index both axes if so desired.

**Why does assignment fail when using chained indexing?**

The problem in the previous section is just a performance issue. What’s up with the SettingWithCopy warning? We don’t usually throw warnings around when you do something that might cost a few extra milliseconds!

But it turns out that assigning to the product of chained indexing has inherently unpredictable results. To see this, think about how the Python interpreter executes this code:

dfmi.loc[:, ('one', 'second')] = value
  # becomes
  dfmi.loc.__setitem__((slice(None), ('one', 'second')), value)

But this code is handled differently:

dfmi['one']['second'] = value
  # becomes
  dfmi.__getitem__('one').__setitem__('second', value)

See that __getitem__ in there? Outside of simple cases, it’s very hard to predict whether it will return a view or a copy (it depends on the memory layout of the array, about which pandas makes no guarantees), and therefore whether the __setitem__ will modify dfmi or a temporary object that gets thrown out immediately afterward. **That's what SettingWithCopy is warning you about!**

**Note:** You may be wondering whether we should be concerned about the loc property in the first example. But

2.5. Indexing and selecting data
dfmi.loc is guaranteed to be dfmi itself with modified indexing behavior, so dfmi.loc.__getitem__/dfmi.loc.__setitem__ operate on dfmi directly. Of course, dfmi.loc.__getitem__(idx) may be a view or a copy of dfmi.

Sometimes a SettingWithCopy warning will arise at times when there’s no obvious chained indexing going on. These are the bugs that SettingWithCopy is designed to catch! pandas is probably trying to warn you that you’ve done this:

```python
def do_something(df):
    foo = df[['bar', 'baz']]  # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...
    # We don't know whether this will modify df or not!
    foo['quux'] = value
    return foo
```

Yikes!

**Evaluation order matters**

When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.

pandas has the SettingWithCopyWarning because assigning to a copy of a slice is frequently not intentional, but a mistake caused by chained indexing returning a copy where a slice was expected.

If you would like pandas to be more or less trusting about assignment to a chained indexing expression, you can set the option mode.chained_assignment to one of these values:

- 'warn', the default, means a SettingWithCopyWarning is printed.
- 'raise' means pandas will raise a SettingWithCopyException you have to deal with.
- None will suppress the warnings entirely.

```python
In [360]: dfb = pd.DataFrame({'a': ['one', 'one', 'two',
                             'three', 'two', 'one', 'six'],
                          'c': np.arange(7)})
....:

# This will show the SettingWithCopyWarning
# but the frame values will be set
In [361]: dfb['c'][dfb['a'].str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```python
>>> pd.set_option('mode.chained_assignment', 'warn')
>>> dfb[dfb['a'].str.startswith('o')][c] = 42
Traceback (most recent call last)
...
SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_index,col_indexer] = value instead
```

A chained assignment can also crop up in setting in a mixed dtype frame.

**Note:** These setting rules apply to all of .loc/.iloc.
The following is the recommended access method using `.loc` for multiple items (using mask) and a single item using a fixed index:

```
In [362]: dfc = pd.DataFrame({'a': ['one', 'one', 'two',
                                  'three', 'two', 'one', 'six'],
                       'c': np.arange(7))

In [363]: dfd = dfc.copy()

# Setting multiple items using a mask
In [364]: mask = dfd['a'].str.startswith('o')

In [365]: dfd.loc[mask, 'c'] = 42

In [366]: dfd
Out[366]:
     a  c
0    one  42
1    one  42
2   two  2
3  three  3
4   two  4
5    one  42
6   six  6

# Setting a single item
In [367]: dfd = dfc.copy()

In [368]: dfd.loc[2, 'a'] = 11

In [369]: dfd
Out[369]:
     a  c
0   one  0
1   one  1
2   11  2
3  three  3
4   two  4
5    one  5
6   six  6
```

The following *can* work at times, but it is not guaranteed to, and therefore should be avoided:

```
In [370]: dfd = dfc.copy()

In [371]: dfd['a'][2] = 111

In [372]: dfd
Out[372]:
     a  c
0   one  0
1   one  1
2  111  2
3  three  3
4   two  4
5    one  5
6   six  6
```

(continues on next page)
Last, the subsequent example will not work at all, and so should be avoided:

```python
>>> pd.set_option('mode.chained_assignment', 'raise')
>>> dfd.loc[0]['a'] = 1111
Traceback (most recent call last)
... SettingWithCopyException:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_index,col_indexer] = value instead
```

**Warning:** The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.

### 2.6 MultIndex / advanced indexing

This section covers indexing with a `MultiIndex` and other advanced indexing features. See the Indexing and Selecting Data for general indexing documentation.

**Warning:** Whether a copy or a reference is returned for a setting operation may depend on the context. This is sometimes called **chained assignment** and should be avoided. See Returning a View versus Copy.

See the cookbook for some advanced strategies.

#### 2.6.1 Hierarchical indexing (MultiIndex)

Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like `Series` (1d) and `DataFrame` (2d).

In this section, we will show what exactly we mean by “hierarchical” indexing and how it integrates with all of the pandas indexing functionality described above and in prior sections. Later, when discussing `group by` and `pivoting and reshaping data`, we’ll show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the cookbook for some advanced strategies.
Creating a MultiIndex (hierarchical index) object

The MultiIndex object is the hierarchical analogue of the standard Index object which typically stores the axis labels in pandas objects. You can think of MultiIndex as an array of tuples where each tuple is unique. A MultiIndex can be created from a list of arrays (using MultiIndex.from_arrays()), an array of tuples (using MultiIndex.from_tuples()), a crossed set of iterables (using MultiIndex.from_product()), or a DataFrame (using MultiIndex.from_frame()). The Index constructor will attempt to return a MultiIndex when it is passed a list of tuples. The following examples demonstrate different ways to initialize MultiIndexes.

1. Creating a MultiIndex from a list of arrays:

```python
arrays = [
    ...:
    ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
    ...:
    ["one", "two", "one", "two", "one", "two", "one", "two"],
    ...: ]
...

tuples = list(zip(*arrays))

In [2]: tuples
...:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

2. Creating a MultiIndex from a DataFrame:

```python
s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]:
first  second
bar one   0.469112
    two  -0.282863
baz one   1.212112
    two -0.173215
foo one   0.469112
    two  -0.282863
```

(continues on next page)
When you want every pairing of the elements in two iterables, it can be easier to use the `MultiIndex.from_product()` method:

```python
In [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]  
In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])  
Out[9]:  
MultiIndex([(  
'bar', 'one'),  
('bar', 'two'),  
('baz', 'one'),  
('baz', 'two'),  
('foo', 'one'),  
('foo', 'two'),  
('qux', 'one'),  
('qux', 'two')],  
names=['first', 'second'])
```

You can also construct a `MultiIndex` from a `DataFrame` directly, using the method `MultiIndex.from_frame()`. This is a complementary method to `MultiIndex.to_frame()`.

```python
In [10]: df = pd.DataFrame(  
....:     [['bar', 'one'], ['bar', 'two'], ['foo', 'one'], ['foo', 'two']],  
....:     columns=['first', 'second'],  
....: )  
....:
In [11]: pd.MultiIndex.from_frame(df)  
Out[11]:  
MultiIndex([(  
'bar', 'one'),  
('bar', 'two'),  
('foo', 'one'),  
('foo', 'two')],  
names=['first', 'second'])
```

As a convenience, you can pass a list of arrays directly into `Series` or `DataFrame` to construct a `MultiIndex` automatically:

```python
In [12]: arrays = [  
....:     np.array(['bar', 'baz', 'foo', 'qux']),  
....:     np.array(['one', 'two', 'one', 'two']),  
....: ]  
....:
In [13]: s = pd.Series(np.random.randn(8), index=arrays)  
In [14]: s  
Out[14]:  
bar one  -0.861849  
two -2.104569
```
All of the MultiIndex constructors accept a names argument which stores string names for the levels themselves. If no names are provided, None will be assigned:

```
In [17]: df.index.names
Out[17]: FrozenList([None, None])
```

This index can back any axis of a pandas object, and the number of levels of the index is up to you:

```
In [18]: df = pd.DataFrame(np.random.randn(3, 8), index="A", "B", "C"); columns=index);

In [19]: df
```

We’ve “sparsified” the higher levels of the indexes to make the console output a bit easier on the eyes. Note that how
the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_options()`:

```
In [21]: with pd.option_context("display.multi_sparse", False):
    .....:   df
    .....:
```

It’s worth keeping in mind that there’s nothing preventing you from using tuples as atomic labels on an axis:

```
In [22]: pd.Series(np.random.randn(8), index=tuples)
Out[22]:
(bar, one) -1.236269
(bar, two)  0.896171
(baz, one) -0.487602
(baz, two) -0.082240
(foo, one) -2.182937
(foo, two)  0.380396
(qux, one)  0.084844
(qux, two)  0.432390
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

Reconstructing the level labels

The method `get_level_values()` will return a vector of the labels for each location at a particular level:

```
In [23]: index.get_level_values(0)
Out[23]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], dtype='object',
            name='first')

In [24]: index.get_level_values("second")
Out[24]: Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'], dtype='object',
            name='second')
```

Basic indexing on axis with MultiIndex

One of the important features of hierarchical indexing is that you can select data by a “partial” label identifying a subgroup in the data. Partial selection “drops” levels of the hierarchical index in the result in a completely analogous way to selecting a column in a regular DataFrame:

```
In [25]: df["bar"]
Out[25]:
second     one     two
 A  0.895717  0.805244
 B  0.410835  0.813850
 C -1.413681  1.607920

In [26]: df["bar", "one"]
Out[26]:
```

(continues on next page)
A 0.895717
B 0.410835
C -1.413681
Name: (bar, one), dtype: float64

In [27]: df["bar"]["one"]
Out[27]:
A 0.895717
B 0.410835
C -1.413681
Name: one, dtype: float64

In [28]: s["qux"]
Out[28]:
one -1.039575
two 0.271860
dtype: float64

See Cross-section with hierarchical index for how to select on a deeper level.

Defined levels

The MultiIndex keeps all the defined levels of an index, even if they are not actually used. When slicing an index, you may notice this. For example:

In [29]: df.columns.levels # original MultiIndex
Out[29]: FrozenList([["bar", "baz", "foo", "qux"], ["one", "two"]])

In [30]: df["foo","qux"].columns.levels # sliced
Out[30]: FrozenList([["bar", "baz", "foo", "qux"], ["one", "two"]])

This is done to avoid a recomputation of the levels in order to make slicing highly performant. If you want to see only the used levels, you can use the get_level_values() method.

In [31]: df["foo", "qux"].columns.to_numpy()
Out[31]:
array([["foo", "one"], ["foo", "two"], ["qux", "one"], ["qux", "two"],
       dtype=object),

# for a specific level
In [32]: df["foo", "qux"].columns.get_level_values(0)
Out[32]: Index(["foo", "foo", "qux", "qux"], dtype=object, name='first')

To reconstruct the MultiIndex with only the used levels, the remove_unused_levels() method may be used.

In [33]: new_mi = df["foo", "qux"].columns.remove_unused_levels()

In [34]: new_mi.levels
Out[34]: FrozenList([["foo", "qux"], ["one", "two"]])
Data alignment and using reindex

Operations between differently-indexed objects having MultiIndex on the axes will work as you expect; data alignment will work the same as an Index of tuples:

```python
In [35]: s + s[1:-2]
Out[35]:
bar one   -1.723698
   two   -4.209138
baz one   -0.989859
   two    2.143608
foo one    1.443110
   two   -1.413542
qux one     NaN
   two     NaN
dtype: float64

In [36]: s + s[1::2]
Out[36]:
bar one   -1.723698
   two    NaN
baz one   -0.989859
   two    NaN
foo one    1.443110
   two    NaN
qux one   -2.079150
   two    NaN
dtype: float64
```

The `reindex()` method of Series/DataFrames can be called with another MultiIndex, or even a list or array of tuples:

```python
In [37]: s.reindex(index[:3])
Out[37]:
   first  second
bar one   -0.861849
   two   -2.104569
baz one   -0.494929
dtype: float64

In [38]: s.reindex([("foo", "two"),("bar", "one"),("qux", "one"), ("baz", "one")])
Out[38]:
   foo  two    -0.706771
bar one   -0.861849
qux one   -1.039575
baz one   -0.494929
dtype: float64
```
2.6.2 Advanced indexing with hierarchical index

Syntactically integrating MultiIndex in advanced indexing with .loc is a bit challenging, but we’ve made every effort to do so. In general, MultiIndex keys take the form of tuples. For example, the following works as you would expect:

```
In [39]: df = df.T

In [40]: df
Out[40]:
   A   B   C
first second
bar  one  0.895717  0.410835 -1.413681
two  0.805244  0.813850  1.607920
baz  one -1.206412  0.132003  1.024180
two  2.565646 -0.827317  0.569605
foo  one  1.431256 -0.076467  0.875906
two  1.340309 -1.187678 -2.211372
qux  one -1.170299  1.130127  0.974466
two  2.226169 -1.436737 -2.006747
```

In [41]: df.loc[('bar', 'two')]

```
Out[41]:
   A       
bar one  0.805244
     two  0.813850
```

Note that df.loc['bar', 'two'] would also work in this example, but this shorthand notation can lead to ambiguity in general.

If you also want to index a specific column with .loc, you must use a tuple like this:

```
In [42]: df.loc[('bar', 'two'), 'A']
Out[42]:
0.805244
```

You don’t have to specify all levels of the MultiIndex by passing only the first elements of the tuple. For example, you can use “partial” indexing to get all elements with bar in the first level as follows:

```
In [43]: df.loc["bar"]
Out[43]:
   A   B   C
second
one  0.895717  0.410835 -1.413681
two  0.805244  0.813850  1.607920
```

This is a shortcut for the slightly more verbose notation df.loc[('bar',)], (equivalent to df.loc['bar',] in this example).

“Partial” slicing also works quite nicely.

```
In [44]: df.loc["baz":"foo"]
Out[44]:
   A   B   C
first second
baz  one -1.206412  0.132003  1.024180
```

(continues on next page)
You can slice with a ‘range’ of values, by providing a slice of tuples.

```
In [45]: df.loc[('baz', 'two'):('qux', 'one')]
Out[45]:
   A    B    C
first  
  baz 2.565646 -0.827317  0.569605
  foo 1.431256 -0.076467  0.875906
  two 1.340309 -1.187678 -2.211372
  qux -1.170299  1.130127  0.974466
```

```
In [46]: df.loc[('baz', 'two'):('foo')]
Out[46]:
   A    B    C
first  
  baz 2.565646 -0.827317  0.569605
  foo 1.431256 -0.076467  0.875906
  two 1.340309 -1.187678 -2.211372
```

Passing a list of labels or tuples works similar to reindexing:

```
In [47]: df.loc[['bar', 'two'], ('qux', 'one')]
Out[47]:
   A    B    C
first  
  bar 0.805244  0.813850  1.607920
  qux -1.170299  1.130127  0.974466
```

Note: It is important to note that tuples and lists are not treated identically in pandas when it comes to indexing. Whereas a tuple is interpreted as one multi-level key, a list is used to specify several keys. Or in other words, tuples go horizontally (traversing levels), lists go vertically (scanning levels).

Importantly, a list of tuples indexes several complete MultiIndex keys, whereas a tuple of lists refer to several values within a level:

```
In [48]: s = pd.Series(...:
    [1, 2, 3, 4, 5, 6],
    index=pd.MultiIndex.from_product([['A', 'B'], ['c', 'd', 'e']]),
    )
    ...
```

```
In [49]: s.loc[['A', 'c'], ('B', 'd')]] # list of tuples
Out[49]:
   A  c  1
   B  d  5
dtype: int64
```
Using slicers

You can slice a MultiIndex by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see Selection by Label, including slices, lists of labels, labels, and boolean indexers.

You can use slice(None) to select all the contents of that level. You do not need to specify all the deeper levels, they will be implied as slice(None).

As usual, both sides of the slicers are included as this is label indexing.

**Warning:** You should specify all axes in the .loc specifier, meaning the indexer for the index and for the columns. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing both axes, rather than into say the MultiIndex for the rows.

You should do this:

```python
df.loc[(slice("A1", "A3"), ...), :]  # noqa: E999
```

You should **not** do this:

```python
df.loc[(slice("A1", "A3"), ...)]  # noqa: E999
```

```python
In [51]: def mklbl(prefix, n):
    ...:     return ["%s%s" % (prefix, i) for i in range(n)]
    ...:

In [52]: miindex = pd.MultiIndex.from_product(
    ...:

In [53]: micolumns = pd.MultiIndex.from_tuples(
    ...:     ["a", "foo"), ("a", "bar"), ("b", "foo"), ("b", "bah")], names=["lvl0", ...
    ...:     "lvl1")
    ...:

In [54]: dfmi = (pd.DataFrame(
    ...:     np.arange(len(miindex) * len(micolumns)).reshape(
    ...:     (len(miindex), len(micolumns))
    ...: ),
```

(continues on next page)
.....:     index=miindex,
.....:     columns=micolumns,
.....:     )
.....:     .sort_index()
.....:     .sort_index(axis=1)
.....:     )

In [55]: dfmi
Out[55]:
lvl0    a  b  
lvl1         bar  foo  bah  foo
A0  B0  C0  D0  1  0  3  2
   D1  5  4  7  6
   C1  D0  9  8 11 10
   D1  13 12 15 14
   C2  D0  17 16 19 18
... ... ... ... ...
A3  B1  C1  D1  237 236 239 238
   C2  D0  241 240 243 242
   D1  245 244 247 246
   C3  D0  249 248 251 250
   D1  253 252 255 254
[64 rows x 4 columns]

Basic MultiIndex slicing using slices, lists, and labels.

In [56]: dfmi.loc[(slice("A1", "A3"), slice(None), ["C1", "C3"]), :]
Out[56]:
lvl0    a  b  
lvl1         bar  foo  bah  foo
A1  B0  C1  D0  73 72  75  74
   D1  77 76  79  78
   C3  D0  89 88  91  90
   D1  93 92  95  94
   B1  C1  D0 105 104 107 106
... ... ... ... ...
A3  B0  C3  D1  221 220 223 222
   B1  C1  D0  233 232 235 234
   D1  237 236 239 238
   C3  D0  249 248 251 250
   D1  253 252 255 254
[24 rows x 4 columns]

You can use pandas.IndexSlice to facilitate a more natural syntax using :, rather than using slice(None).

In [57]: idx = pd.IndexSlice

In [58]: dfmi.loc[idx[:, :, ["C1", "C3"]], idx[:, "foo"]]
Out[58]:

(continues on next page)
It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```python
In [59]: dfmi.loc["A1", (slice(None), "foo")]
Out[59]:
lvl0   a    b
lvl1  foo  foo
A0  B0  C1  D0  8    10
    D1  12   14
    C3  D0  24   26
    D1  28   30
B1  C1  D0  40   42
...
...
...
A3  B0  C3  D1  220  222
B1  C1  D0  232  234
    D1  236  238
    C3  D0  248  250
    D1  252  254
[16 rows x 2 columns]
```

```python
In [60]: dfmi.loc[idx[:, :, ["C1", "C3"]], idx[:, "foo"]]
Out[60]:
lvl0   a    b
lvl1  foo  foo
A0  B0  C1  D0  8    10
    D1  12   14
    C3  D0  24   26
    D1  28   30
B1  C1  D0  40   42
...
...
...
A3  B0  C3  D1  220  222
B1  C1  D0  232  234
    D1  236  238
    C3  D0  248  250
    D1  252  254
```

(continues on next page)
Using a boolean indexer you can provide selection related to the values.

```python
In [61]: mask = dfmi[('a', 'foo')] > 200

In [62]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[62]:
        a b
   lvl0  
   lvl1    foo  foo
   A3  B0  C1  D1  204  206
   C3  D0  216  218
   D1  220  222
   B1  C1  D0  232  234
   D1  236  238
   C3  D0  248  250
   D1  252  254
```

You can also specify the axis argument to .loc to interpret the passed slicers on a single axis.

```python
In [63]: dfmi.loc(axis=0)[:, :, ['C1', 'C3']]
Out[63]:
    a  b
   lvl0  
   lvl1    bar  foo  bah  foo
   A0  B0  C1  D0   9   8   11  10
   D1   13  12  15  14
   C3  D0   25  24  27  26
   D1   29  28  31  30
   B1  C1  D0   41  40  43  42
   ... ... ... ... ...
   A3  B0  C3  D1  221  220  223  222
   B1  C1  D0  233  232  235  234
   D1  237  236  239  238
   C3  D0  249  248  251  250
   D1  253  252  255  254
```

Furthermore, you can set the values using the following methods.

```python
In [64]: df2 = dfmi.copy()

In [65]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

In [66]: df2
Out[66]:
    a  b
   lvl0  
   lvl1    bar  foo  bah  foo
   A0  B0  C0  D0    1    0    3    2
   D1    5    4    7    6
   C1  D0   -10  -10  -10  -10
   D1   -10  -10  -10  -10
   C2  D0    17    16    19    18
```

(continues on next page)
You can use a right-hand-side of an alignable object as well.

```python
In [67]: df2 = dfmi.copy()

In [68]: df2.loc[idx[:, :, ["C1", "C3"]], :] = df2 * 1000

In [69]: df2
```

```python
Out[69]:
```

Cross-section

The `xs()` method of DataFrame additionally takes a level argument to make selecting data at a particular level of a MultiIndex easier.

```python
In [70]: df
```

```python
Out[70]:
```

2.6. MultiIndex / advanced indexing
In [71]: df.xs("one", level="second")
Out[71]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>0.895717</td>
<td>0.410835</td>
<td>-1.413681</td>
</tr>
<tr>
<td>baz</td>
<td>-1.206412</td>
<td>0.132003</td>
<td>1.024180</td>
</tr>
<tr>
<td>foo</td>
<td>1.431256</td>
<td>-0.076467</td>
<td>0.875906</td>
</tr>
<tr>
<td>qux</td>
<td>-1.170299</td>
<td>1.130127</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

# using the slicers

In [72]: df.loc[(slice(None), "one"), :]
Out[72]:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>0.895717</td>
<td>0.410835</td>
<td>-1.413681</td>
</tr>
<tr>
<td>baz</td>
<td>-1.206412</td>
<td>0.132003</td>
<td>1.024180</td>
</tr>
<tr>
<td>foo</td>
<td>1.431256</td>
<td>-0.076467</td>
<td>0.875906</td>
</tr>
<tr>
<td>qux</td>
<td>-1.170299</td>
<td>1.130127</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

You can also select on the columns with `xs`, by providing the axis argument.

In [73]: df = df.T
In [74]: df.xs("one", level="second", axis=1)
Out[74]:
<table>
<thead>
<tr>
<th>first</th>
<th>bar</th>
<th>baz</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.895717</td>
<td>-1.206412</td>
<td>1.431256</td>
<td>-1.170299</td>
</tr>
<tr>
<td>B</td>
<td>0.410835</td>
<td>0.132003</td>
<td>-0.076467</td>
<td>1.130127</td>
</tr>
<tr>
<td>C</td>
<td>-1.413681</td>
<td>1.024180</td>
<td>0.875906</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

# using the slicers

In [75]: df.loc[:, (slice(None), "one")]
Out[75]:
<table>
<thead>
<tr>
<th>first</th>
<th>bar</th>
<th>baz</th>
<th>foo</th>
<th>qux</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.895717</td>
<td>-1.206412</td>
<td>1.431256</td>
<td>-1.170299</td>
</tr>
<tr>
<td>B</td>
<td>0.410835</td>
<td>0.132003</td>
<td>-0.076467</td>
<td>1.130127</td>
</tr>
<tr>
<td>C</td>
<td>-1.413681</td>
<td>1.024180</td>
<td>0.875906</td>
<td>0.974466</td>
</tr>
</tbody>
</table>

xs also allows selection with multiple keys.

In [76]: df.xs(("one", "bar"), level=("second", "first"), axis=1)
Out[76]:
<table>
<thead>
<tr>
<th>first</th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.895717</td>
</tr>
<tr>
<td>B</td>
<td>0.410835</td>
</tr>
<tr>
<td>C</td>
<td>-1.413681</td>
</tr>
</tbody>
</table>

# using the slicers

In [77]: df.loc[:, ("bar", "one")]
Out[77]:
(continues on next page)
You can pass `drop_level=False` to `xs` to retain the level that was selected.

```
In [78]: df.xs("one", level="second", axis=1, drop_level=False)
Out[78]:
      first     bar     baz     foo     qux
second one     one     one     one     one
      A  0.895717 -1.206412  1.431256 -1.170299
      B  0.410835  0.132003 -0.076467  1.130127
      C -1.413681  1.024180  0.875906  0.974466
```

Compare the above with the result using `drop_level=True` (the default value).

```
In [79]: df.xs("one", level="second", axis=1, drop_level=True)
Out[79]:
      first     bar     baz     foo     qux
      A  0.895717 -1.206412  1.431256 -1.170299
      B  0.410835  0.132003 -0.076467  1.130127
      C -1.413681  1.024180  0.875906  0.974466
```

### Advanced reindexing and alignment

Using the parameter `level` in the `reindex()` and `align()` methods of pandas objects is useful to broadcast values across a level. For instance:

```
In [80]: midx = pd.MultiIndex(  
    ...:     levels=["zero", "one"], ["x", "y"], codes=[[1, 1, 0, 0], [1, 0, 1, 0]])
In [81]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)
In [82]: df
Out[82]:
0 1
one y 1.519970 -0.493662
  x 0.600178  0.274230
zero y 0.132885 -0.023688
  x 2.410179  1.450520
```

```
In [83]: df2 = df.groupby(level=0).mean()
In [84]: df2
Out[84]:
0 1
one 1.060074 -0.109716
  1.271532  0.713416
```
In [85]: df2.reindex(df.index, level=0)
Out[85]:
    0  1
one y 1.060074 -0.109716
  x 1.060074 -0.109716
zero y 1.271532  0.713416
  x 1.271532  0.713416

# aligning
In [86]: df_aligned, df2_aligned = df.align(df2, level=0)

In [87]: df_aligned
Out[87]:
   0  1
one y 1.519970 -0.493662
  x  0.600178  0.274230
zero y 0.132885 -0.023688
  x  2.410179  1.450520

In [88]: df2_aligned
Out[88]:
   0  1
one y 1.060074 -0.109716
  x 1.060074 -0.109716
zero y 1.271532  0.713416
  x 1.271532  0.713416

Swapping levels with swaplevel

The `swaplevel()` method can switch the order of two levels:

In [89]: df[:5]
Out[89]:
   0  1
one y 1.519970 -0.493662
  x  0.600178  0.274230
zero y 0.132885 -0.023688
  x  2.410179  1.450520

In [90]: df[:5].swaplevel(0, 1, axis=0)
Out[90]:
   0  1
  y one 1.519970 -0.493662
  x one  0.600178  0.274230
  y zero 0.132885 -0.023688
  x zero  2.410179  1.450520
Reordering levels with `reorder_levels`

The `reorder_levels()` method generalizes the `swaplevel` method, allowing you to permute the hierarchical index levels in one step:

```
In [91]: df[:5].reorder_levels([1, 0], axis=0)
Out[91]:
     0    1
y one  1.519970 -0.493662
x one  0.600178  0.274230
y zero 0.132885 -0.023688
x zero 2.410179  1.450520
```

Renaming names of an Index or MultiIndex

The `rename()` method is used to rename the labels of a `MultiIndex`, and is typically used to rename the columns of a `DataFrame`. The `columns` argument of `rename` allows a dictionary to be specified that includes only the columns you wish to rename.

```
In [92]: df.rename(columns={0: "col0", 1: "col1"})
Out[92]:
     col0   col1
one y  1.519970 -0.493662
x     0.600178  0.274230
zero y 0.132885 -0.023688
x      2.410179  1.450520
```

This method can also be used to rename specific labels of the main index of the `DataFrame`.

```
In [93]: df.rename(index={"one": "two", "y": "z"})
Out[93]:
     0    1
two z  1.519970 -0.493662
x     0.600178  0.274230
zero z 0.132885 -0.023688
x     2.410179  1.450520
```

The `rename_axis()` method is used to rename the name of a Index or MultiIndex. In particular, the names of the levels of a `MultiIndex` can be specified, which is useful if `reset_index()` is later used to move the values from the `MultiIndex` to a column.

```
In [94]: df.rename_axis(index=["abc", "def"])
Out[94]:
     0    1
abc def
one y  1.519970 -0.493662
x     0.600178  0.274230
zero y 0.132885 -0.023688
x     2.410179  1.450520
```

Note that the columns of a `DataFrame` are an index, so that using `rename_axis` with the `columns` argument will change the name of that index.
Both `rename` and `rename_axis` support specifying a dictionary, Series or a mapping function to map labels/names to new values.

When working with an `Index` object directly, rather than via a `DataFrame`, `Index.set_names()` can be used to change the names.

```python
In [96]: mi = pd.MultiIndex.from_product([[1, 2], ['a', 'b']], names=['x', 'y'])
In [97]: mi.names
Out[97]: FrozenList(['x', 'y'])
In [98]: mi2 = mi.rename("new name", level=0)
In [99]: mi2
Out[99]: MultiIndex([(1,'a'),
                   (1,'b'),
                   (2,'a'),
                   (2,'b')],
                  names=['new name', 'y'])
```

You cannot set the names of the MultiIndex via a level.

```python
In [100]: mi.levels[0].name = "name via level"
```

```
RuntimeError: Cannot set name on a level of a MultiIndex. Use 'MultiIndex.set_names' instead.
```

Use `Index.set_names()` instead.
2.6.3 Sorting a MultiIndex

For MultiIndex-ed objects to be indexed and sliced effectively, they need to be sorted. As with any index, you can use `sort_index()`.

```python
In [101]: import random

In [102]: random.shuffle(tuples)

In [103]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))

In [104]: s
Out[104]:

    baz  one  0.206053
        two  1.063327
    foo  one  0.251905
        two  1.266143
    qux  one  2.213588
        two  0.408204
    bar  one  0.299368
        two  0.863838
    dtype: float64

In [105]: s.sort_index()
Out[105]:

    bar  one  0.299368
        two  1.266143
    baz  one  0.206053
        two  0.408204
    foo  one  0.251905
        two  0.863838
    qux  one  2.213588
        two  1.063327
    dtype: float64

In [106]: s.sort_index(level=0)
Out[106]:

    bar  one  0.299368
        two  1.266143
    baz  one  0.206053
        two  0.408204
    foo  one  0.251905
        two  0.863838
    qux  one  2.213588
        two  1.063327
    dtype: float64

In [107]: s.sort_index(level=1)
Out[107]:

    bar  one  0.299368
    baz  one  0.206053
    foo  one  0.251905
    qux  one  2.213588
```

(continues on next page)
You may also pass a level name to \texttt{sort.index} if the \texttt{MultiIndex} levels are named.

\begin{verbatim}
In [108]: s.index.set_names(["L1", "L2"], inplace=True)

In [109]: s.sort_index(level="L1")
Out[109]:
L1 L2
bar one  0.299368
two  1.266143
baz one  0.206053
two  0.408204
foo one  -0.251905
two  -0.863838
qux one  -2.213588
two  1.063327
dtype: float64

In [110]: s.sort_index(level="L2")
Out[110]:
L1 L2
bar one  0.299368
baz one  0.206053
foo one  -0.251905
qux one  -2.213588
bar two  1.266143
baz two  0.408204
foo two  -0.863838
qux two  1.063327
dtype: float64
\end{verbatim}

On higher dimensional objects, you can sort any of the other axes by level if they have a \texttt{MultiIndex}:

\begin{verbatim}
In [111]: df.T.sort_index(level=1, axis=1)
Out[111]:
       one  zero  one  zero
       x     x     y     y
    0  0.600178  2.410179  1.519970  0.132885
    1  0.274230  1.450520 -0.493662 -0.023688

In [112]: dfm = pd.DataFrame(
    ...:     .....:     {"jim": [0, 0, 1, 1], "joe": ["x", "x", "z", "y"], "jolie": np.random.rand(4)
    ...:     .....:     }
    .....:     )
\end{verbatim}
In [113]: dfm = dfm.set_index(['jim', 'joe'])

In [114]: dfm
Out[114]:
jolie
jim joe
0 x 0.490671
 x 0.120248
l y 0.110968
 z 0.537020

In [4]: dfm.loc[(1,'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.
Out[4]:
jolie
jim joe
l z 0.64094

Furthermore, if you try to index something that is not fully lexsorted, this can raise:

In [5]: dfm.loc[(0,'y'):(1,'z')]
UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'

The is_monotonic_increasing() method on a MultiIndex shows if the index is sorted:

In [115]: dfm.index.is_monotonic_increasing
Out[115]: False

In [116]: dfm = dfm.sort_index()

In [117]: dfm
Out[117]:
jolie
jim joe
0 x 0.490671
 x 0.120248
l y 0.110968
 z 0.537020

In [118]: dfm.index.is_monotonic_increasing
Out[118]: True

And now selection works as expected.

In [119]: dfm.loc[(0,'y'):(1,'z')]
Out[119]:
jolie
jim joe
l y 0.110968
 z 0.537020
2.6.4 Take methods

Similar to NumPy ndarrays, pandas Index, Series, and DataFrame also provides the `take()` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [120]: index = pd.Index(np.random.randint(0, 1000, 10))
In [121]: index
Out[121]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64')
In [122]: positions = [0, 9, 3]
In [123]: index[positions]
Out[123]: Int64Index([214, 329, 567], dtype='int64')
In [124]: index.take(positions)
Out[124]: Int64Index([214, 329, 567], dtype='int64')
In [125]: ser = pd.Series(np.random.randn(10))
In [126]: ser.iloc[positions]
Out[126]:
0  -0.179666
9   1.824375
3   0.392149
dtype: float64
In [127]: ser.take(positions)
Out[127]:
0  -0.179666
9   1.824375
3   0.392149
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [128]: frm = pd.DataFrame(np.random.randn(5, 3))
In [129]: frm.take([1, 4, 3])
Out[129]:
          0    1    2
0  1.042766 0.330937 -1.730291
1 -1.237881 0.106854 -1.276829
4  0.629675 -1.425966  1.857704
3  0.979542 -1.633678  0.615855
In [130]: frm.take([0, 2], axis=1)
Out[130]:
          0    2
0  0.595974 0.601544
1 -1.237881 -1.276829
2 -0.767101  1.499591
3  0.979542  0.615855
4  0.629675  1.857704
```
It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```
In [131]: arr = np.random.randn(10)

In [132]: arr.take([False, False, True, True])
Out[132]: array([-1.1935, -1.1935,  0.6775,  0.6775])

In [133]: arr[[0, 1]]
Out[133]: array([-1.1935,  0.6775])

In [134]: ser = pd.Series(np.random.randn(10))

In [135]: ser.take([False, False, True, True])
Out[135]:
   0    0.233141
   1   -0.223540
   dtype: float64

In [136]: ser.iloc[[0, 1]]
Out[136]:
   0    0.233141
   1   -0.223540
dtype: float64
```

Finally, as a small note on performance, because the `take` method handles a narrower range of inputs, it can offer performance that is a good deal faster than fancy indexing.

```
In [137]: arr = np.random.randn(10000, 5)

In [138]: indexer = np.arange(10000)

In [139]: random.shuffle(indexer)

In [140]: %timeit arr[indexer]
       ....: %timeit arr.take(indexer, axis=0)
       ....:
140 ns +- 2.63 ns per loop (mean +- std. dev. of 7 runs, 10,000 loops each)
46.4 ns +- 262 ns per loop (mean +- std. dev. of 7 runs, 10,000 loops each)

In [141]: ser = pd.Series(arr[:, 0])

In [142]: %timeit ser.iloc[indexer]
       ....: %timeit ser.take(indexer)
       ....:
60.7 ns +- 1.2 us per loop (mean +- std. dev. of 7 runs, 10,000 loops each)
53.2 ns +- 1.36 us per loop (mean +- std. dev. of 7 runs, 10,000 loops each)
```
2.6.5 Index types

We have discussed MultiIndex in the previous sections pretty extensively. Documentation about DatetimeIndex and PeriodIndex are shown here, and documentation about TimedeltaIndex is found here.

In the following sub-sections we will highlight some other index types.

CategoricalIndex

CategoricalIndex is a type of index that is useful for supporting indexing with duplicates. This is a container around a Categorical and allows efficient indexing and storage of an index with a large number of duplicated elements.

```python
In [143]: from pandas.api.types import CategoricalDtype
In [144]: df = pd.DataFrame({'A': np.arange(6), 'B': list('aabbca'))
In [145]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))
In [146]: df
Out[146]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a
In [147]: df.dtypes
Out[147]:
A    int64
B    category
dtype: object
In [148]: df['B'].cat.categories
Out[148]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a CategoricalIndex.

```python
In [149]: df2 = df.set_index('B')
In [150]: df2.index
Out[150]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'], ordered=False, dtype='category', name='B')
```

Indexing with __getitem__/.iloc/.loc works similarly to an Index with duplicates. The indexers must be in the category or the operation will raise a KeyError.

```python
In [151]: df2.loc['a']
Out[151]:
   A  B
0  a
```

(continues on next page)
The `CategoricalIndex` is **preserved** after indexing:

```python
In [152]: df2.loc["a"].index
Out[152]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False, dtype='category', name='B')
```

Sorting the index will sort by the order of the categories (recall that we created the index with `CategoricalDtype(list('cab'))`, so the sorted order is `cab`).

```python
In [153]: df2.sort_index()
Out[153]:
   A   B
   c  4
   a  0
   a  1
   a  5
   b  2
   b  3
```

Groupby operations on the index will preserve the index nature as well.

```python
In [154]: df2.groupby(level=0).sum()
Out[154]:
   A   B
   c  4
   a  6
   b  5
```

```python
In [155]: df2.groupby(level=0).sum().index
Out[155]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False, dtype='category', name='B')
```

Reindexing operations will return a resulting index based on the type of the passed indexer. Passing a list will return a plain-old `Index`; indexing with a `Categorical` will return a `CategoricalIndex`, indexed according to the categories of the passed `Categorical` dtype. This allows one to arbitrarily index these even with values **not** in the categories, similarly to how you can reindex any pandas index.

```python
In [156]: df3 = pd.DataFrame({"A": np.arange(3), "B": pd.Series(list("abc"))_.astype("category")})
In [157]: df3 = df3.set_index("B")
In [158]: df3
Out[158]:
   A
   B
```

(continues on next page)
In [159]: df3.reindex(["a", "e"])
Out[159]:
   A  
   B  
   a  0.0
   e  NaN

In [160]: df3.reindex(["a", "e"]).index
Out[160]: Index(["a", "e"], dtype='object', name='B')

In [161]: df3.reindex(pd.Categorical(["a", "e"], categories=list("abe")))
Out[161]:
   A  
   B  
   a  0.0
   e  NaN

In [162]: df3.reindex(pd.Categorical(["a", "e"], categories=list("abe"))).index
Out[162]: CategoricalIndex(["a", "e"], categories=["a", "b", "e"], ordered=False, dtype='category', name='B')

Warning: Reshaping and Comparison operations on a CategoricalIndex must have the same categories or a TypeError will be raised.

In [163]: df4 = pd.DataFrame({"A": np.arange(2), "B": list("ba")})
In [164]: df4["B"] = df4["B"].astype(CategoricalDtype(list("ab")))
In [165]: df4 = df4.set_index("B")
Out[165]: CategoricalIndex(["b", "a"], categories=["a", "b"], ordered=False, dtype='category', name='B')

In [166]: df4.index
Out[166]: CategoricalIndex(["b", "a"], categories=["a", "b"], ordered=False, dtype='category', name='B')

In [167]: df5 = pd.DataFrame({"A": np.arange(2), "B": list("bc")})
In [168]: df5["B"] = df5["B"].astype(CategoricalDtype(list("bc")))
In [169]: df5 = df5.set_index("B")
Out[169]: CategoricalIndex(["b", "c"], categories=["b", "c"], ordered=False, dtype='category', name='B')

In [170]: df5.index
Out[170]: CategoricalIndex(["b", "c"], categories=["b", "c"], ordered=False, dtype='category', name='B')

In [1]: pd.concat([df4, df5])
Type Error: categories must match existing categories when appending
Int64Index and RangeIndex

Deprecated since version 1.4.0: In pandas 2.0, Index will become the default index type for numeric types instead of Int64Index, Float64Index and UInt64Index and those index types are therefore deprecated and will be removed in a future version. RangeIndex will not be removed, as it represents an optimized version of an integer index.

Int64Index is a fundamental basic index in pandas. This is an immutable array implementing an ordered, sliceable set.

RangeIndex is a sub-class of Int64Index that provides the default index for all NDFrame objects. RangeIndex is an optimized version of Int64Index that can represent a monotonic ordered set. These are analogous to Python range types.

Float64Index

Deprecated since version 1.4.0: Index will become the default index type for numeric types in the future instead of Int64Index, Float64Index and UInt64Index and those index types are therefore deprecated and will be removed in a future version of Pandas. RangeIndex will not be removed as it represents an optimized version of an integer index.

By default a Float64Index will be automatically created when passing floating, or mixed-integer-floating values in index creation. This enables a pure label-based slicing paradigm that makes [],ix,loc for scalar indexing and slicing work exactly the same.

```
In [171]: indexf = pd.Index([1.5, 2, 3, 4.5, 5])

In [172]: indexf
Out[172]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [173]: sf = pd.Series(range(5), index=indexf)

In [174]: sf
Out[174]:
1.5  0
2.0  1
3.0  2
4.5  3
5.0  4
dtype: int64
```

Scalar selection for [], .loc will always be label based. An integer will match an equal float index (e.g. 3 is equivalent to 3.0).

```
In [175]: sf[3]
Out[175]: 2

In [176]: sf[3.0]
Out[176]: 2

In [177]: sf.loc[3]
Out[177]: 2

In [178]: sf.loc[3.0]
Out[178]: 2
```

2.6. MultiIndex / advanced indexing
The only positional indexing is via \texttt{iloc}.

```python
In [179]: sf.iloc[3]
Out[179]: 3
```

A scalar index that is not found will raise a \texttt{KeyError}. Slicing is primarily on the values of the index when using [], \texttt{ix}, \texttt{loc}, and \texttt{always} positional when using \texttt{iloc}. The exception is when the slice is boolean, in which case it will always be positional.

```python
In [180]: sf[2:4]
Out[180]:
2.0  1
3.0  2
dtype: int64

In [181]: sf.loc[2:4]
Out[181]:
2.0  1
3.0  2
dtype: int64

In [182]: sf.iloc[2:4]
Out[182]:
3.0  2
4.5  3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```python
In [183]: sf[2.1:4.6]
Out[183]:
3.0  2
4.5  3
dtype: int64

In [184]: sf.loc[2.1:4.6]
Out[184]:
3.0  2
4.5  3
dtype: int64
```

In non-float indexes, slicing using floats will raise a \texttt{TypeError}.

```python
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type (Int64Index)
```

Here is a typical use-case for using this type of indexing. Imagine that you have a somewhat irregular timedelta-like indexing scheme, but the data is recorded as floats. This could, for example, be millisecond offsets.

```python
In [185]: dfir = pd.concat(
    .....:     ...
```

(continues on next page)
Selection operations then will always work on a value basis, for all selection operators.

In [187]: dfir[0:1000.4]
Out[187]:
   A     B
0.0 -0.435772 -1.188928
250.0 -0.808286 -0.284634
500.0 -1.815703  1.347213
750.0 -0.243487  0.514704
1000.0  1.162969 -0.287725
1000.4 -0.179734  0.993962

In [188]: dfir.loc[0:1001, "A"]
Out[188]:
   A
0.0 -0.435772
250.0 -0.808286
500.0 -1.815703
750.0 -0.243487
1000.0  1.162969
1000.4 -0.179734
Name: A, dtype: float64

In [189]: dfir.loc[1000.4]
Out[189]:
(continues on next page)
You could retrieve the first 1 second (1000 ms) of data as such:

```
In [190]: dfir[0:1000]
Out[190]:
       A       B
0  -0.435772 -1.188928
250  -0.808286 -0.284634
500  -1.815703  1.347213
750  -0.243487  0.514704
1000  1.162969 -0.287725
```

If you need integer based selection, you should use `iloc`:

```
In [191]: dfir.iloc[0:5]
Out[191]:
       A       B
0  -0.435772 -1.188928
250  -0.808286 -0.284634
500  -1.815703  1.347213
750  -0.243487  0.514704
1000  1.162969 -0.287725
```

### IntervalIndex

`IntervalIndex` together with its own dtype, `IntervalDtype` as well as the `Interval` scalar type, allow first-class support in pandas for interval notation.

The `IntervalIndex` allows some unique indexing and is also used as a return type for the categories in `cut()` and `qcut()`.

#### Indexing with an IntervalIndex

An `IntervalIndex` can be used in `Series` and in `DataFrame` as the index.

```
In [192]: df = pd.DataFrame({
         "A": [1, 2, 3, 4],
         ....:     index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4])
         ....: })
```

```
In [193]: df
Out[193]:
       A
(0, 1]  1
(1, 2]  2
(2, 3]  3
(3, 4]  4
```
Label based indexing via `.loc` along the edges of an interval works as you would expect, selecting that particular interval.

```
In [194]: df.loc[2]
Out[194]:
   A
Name: (1, 2], dtype: int64

In [195]: df.loc[[2, 3]]
Out[195]:
   A
(1, 2] 2
(2, 3] 3
```

If you select a label *contained* within an interval, this will also select the interval.

```
In [196]: df.loc[2.5]
Out[196]:
   A
Name: (2, 3], dtype: int64

In [197]: df.loc[[2.5, 3.5]]
Out[197]:
   A
(2, 3] 3
(3, 4] 4
```

Selecting using an Interval will only return exact matches (starting from pandas 0.25.0).

```
In [198]: df.loc[pd.Interval(1, 2)]
Out[198]:
   A
Name: (1, 2], dtype: int64

In [7]: df.loc[pd.Interval(0.5, 2.5)]
---------------------------------------------------------------------------
KeyError: Interval(0.5, 2.5, closed='right')
```

Selecting all Intervals that overlap a given Interval can be performed using the `overlaps()` method to create a boolean indexer.

```
In [199]: idxr = df.index.overlaps(pd.Interval(0.5, 2.5))
In [200]: idxr
Out[200]: array([ True, True, True, False])
In [201]: df[idxr]
Out[201]:
   A
(0, 1] 1
(1, 2] 2
(2, 3] 3
```

2.6. MultiIndex / advanced indexing
Binning data with `cut` and `qcut`

`cut()` and `qcut()` both return a `Categorical` object, and the bins they create are stored as an `IntervalIndex` in its `.categories` attribute.

```python
In [202]: c = pd.cut(range(4), bins=2)
```

```python
In [203]: c
```

```python
Out[203]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]
```

```python
In [204]: c.categories
```

```python
Out[204]: IntervalIndex([(-0.003, 1.5], (1.5, 3.0]], dtype='interval[float64, right]')
```

`cut()` also accepts an `IntervalIndex` for its `bins` argument, which enables a useful pandas idiom. First, we call `cut()` with some data and `bins` set to a fixed number, to generate the bins. Then, we pass the values of `.categories` as the `bins` argument in subsequent calls to `cut()`, supplying new data which will be binned into the same bins.

```python
In [205]: pd.cut([0, 3, 5, 1], bins=c.categories)
```

```python
Out[205]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]
```

Any value which falls outside all bins will be assigned a `NaN` value.

Generating ranges of intervals

If we need intervals on a regular frequency, we can use the `interval_range()` function to create an `IntervalIndex` using various combinations of `start`, `end`, and `periods`. The default frequency for `interval_range` is a 1 for numeric intervals, and calendar day for datetime-like intervals:

```python
In [206]: pd.interval_range(start=0, end=5)
```

```python
Out[206]: IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]], dtype='interval[int64, right]')
```

```python
In [207]: pd.interval_range(start=pd.Timestamp("2017-01-01"), periods=4)
```

```python
Out[207]: IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-01-22], (2017-01-22, 2017-01-29]], dtype='interval[datetime64[ns], right]')
```

The `freq` parameter can be used to specify non-default frequencies, and can utilize a variety of `frequency aliases` with datetime-like intervals:

```python
In [209]: pd.interval_range(start=0, periods=5, freq=1.5)
```

```python
Out[209]: IntervalIndex([(0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]], dtype='interval[float64, right]')
```

```python
In [210]: pd.interval_range(start=pd.Timestamp("2017-01-01"), periods=4, freq="W")
```

```python
Out[210]: IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-01-22], (2017-01-22, 2017-01-29]], dtype='interval[datetime64[ns], right]')
```
In [211]: pd.interval_range(start=pd.Timedelta("0 days"), periods=3, freq="9H")
Out[211]: IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days 18:...
...
1 days 03:00:00]], dtype='interval[timedelta64[ns], right]')

Additionally, the closed parameter can be used to specify which side(s) the intervals are closed on. Intervals are
closed on the right side by default.

In [212]: pd.interval_range(start=0, end=4, closed="both")
Out[212]: IntervalIndex([(0, 1], [1, 2], [2, 3], [3, 4]], dtype='interval[int64, both]')

In [213]: pd.interval_range(start=0, end=4, closed="neither")
Out[213]: IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4]], dtype='interval[int64, neither]')

Specifying start, end, and periods will generate a range of evenly spaced intervals from start to end inclusively,
with periods number of elements in the resulting IntervalIndex:

In [214]: pd.interval_range(start=0, end=6, periods=4)
Out[214]: IntervalIndex([(0.0, 1.5]], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]], dtype='interval[float64, right]')

In [215]: pd.interval_range(pd.Timestamp("2018-01-01"), pd.Timestamp("2018-02-28"),
   periods=3)
Out[215]: IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-
   02-08 16:00:00], (2018-02-08 16:00:00, 2018-02-28]], dtype='interval[datetime64[ns],
   right]')

2.6.6 Miscellaneous indexing FAQ

Integer indexing

Label-based indexing with integer axis labels is a thorny topic. It has been discussed heavily on mailing lists and among
various members of the scientific Python community. In pandas, our general viewpoint is that labels matter more than
integer locations. Therefore, with an integer axis index only label-based indexing is possible with the standard tools
like .loc. The following code will generate exceptions:

In [216]: s = pd.Series(range(5))
In [217]: s[-1]
ValueError Traceback (most recent call last)
File /pandas/pandas/core/indexes/range.py:385, in RangeIndex.get_loc(self, key, method,,
   __tolerance)
   384 try:
   --> 385 return self._range.index(new_key)
   386 except ValueError as err:
ValueError: -1 is not in range

The above exception was the direct cause of the following exception:
KeyError Traceback (most recent call last)
In [217], in <cell line: 1>()
    1 s[-1]
----> 2 File /pandas/pandas/core/series.py:958, in Series.__getitem__(self, key)
       955     return self._values[key]
       957   elif key_is_scalar:
 --> 958     return self._get_value(key)
       960     if is_hashable(key):
       961     # Otherwise index.get_value will raise InvalidIndexError
       962     try:
       963     # For labels that don't resolve as scalars like tuples and frozensets

File /pandas/pandas/core/series.py:1069, in Series._get_value(self, label, takeable)
      1066     return self._values[label]
      1068     # Similar to Index.get_value, but we do not fall back to positional
 --> 1069     loc = self.index.get_loc(label)
      1070     return self.index._get_values_for_loc(self, loc, label)

File /pandas/pandas/core/indexes/range.py:387, in RangeIndex.get_loc(self, key, method, ___tolerance)
     385     return self._range.index(new_key)
     386     except ValueError as err:
 --> 387     raise KeyError(key) from err
     388     self._check_indexing_error(key)
     389     raise KeyError(key)

KeyError: -1

In [218]: df = pd.DataFrame(np.random.randn(5, 4))

In [219]: df
Out[219]:
       0   1   2   3
0 -0.130121 -0.476046 0.759104 0.213379
1 -0.082641 0.448008 0.656420 -1.051443
2  0.594956 -0.151360 -0.069303 1.221431
3 -0.182832 0.791235 0.042745 2.069775
4  1.446552 0.019814 -1.389212 -0.702312

In [220]: df.loc[-2:]
Out[220]:
       0   1   2   3
0 -0.130121 -0.476046 0.759104 0.213379
1 -0.082641 0.448008 0.656420 -1.051443
2  0.594956 -0.151360 -0.069303 1.221431
3 -0.182832 0.791235 0.042745 2.069775
4  1.446552 0.019814 -1.389212 -0.702312

This deliberate decision was made to prevent ambiguities and subtle bugs (many users reported finding bugs when the API change was made to stop “falling back” on position-based indexing).
Non-monotonic indexes require exact matches

If the index of a Series or DataFrame is monotonically increasing or decreasing, then the bounds of a label-based slice can be outside the range of the index, much like slice indexing a normal Python list. Monotonicity of an index can be tested with the `is_monotonic_increasing()` and `is_monotonic_decreasing()` attributes.

```python
In [221]: df = pd.DataFrame(index=[2, 3, 4, 5], columns=['data'], data=list(range(5)))

In [222]: df.index.is_monotonic_increasing
Out[222]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [223]: df.loc[0:4, :]
Out[223]:
   data
2   0
3   1
3   2
4   3

# slice is are outside the index, so empty DataFrame is returned
In [224]: df.loc[13:15, :]
Out[224]:
Empty DataFrame
Columns: [data]
Index: []
```

On the other hand, if the index is not monotonic, then both slice bounds must be unique members of the index.

```python
In [225]: df = pd.DataFrame(index=[2, 3, 1, 4, 3, 5], columns=['data'], data=list(range(6)))

In [226]: df.index.is_monotonic_increasing
Out[226]: False

# OK because 2 and 4 are in the index
In [227]: df.loc[2:4, :]
Out[227]:
   data
2   0
3   1
1   2
4   3

# 0 is not in the index
In [9]: df.loc[0:4, :]
KeyError: 0

# 3 is not a unique label
In [11]: df.loc[2:3, :]
KeyError: 'Cannot get right slice bound for non-unique label: 3'
```

Index.is_monotonic_increasing and Index.is_monotonic_decreasing only check that an index is weakly monotonic. To check for strict monotonicity, you can combine one of those with the `is_unique()` attribute.
Endpoints are inclusive

Compared with standard Python sequence slicing in which the slice endpoint is not inclusive, label-based slicing in pandas is inclusive. The primary reason for this is that it is often not possible to easily determine the “successor” or next element after a particular label in an index. For example, consider the following Series:

```
In [232]: s = pd.Series(np.random.randn(6), index=list("abcdef"))
In [233]: s
Out[233]:
     a     b     c     d     e     f
  0  0.30  1.24 -0.85 -0.04 -1.66 -0.82
dtype: float64
```

Suppose we wished to slice from c to e, using integers this would be accomplished as such:

```
In [234]: s[2:5]
Out[234]:
     c     d     e
  0 -0.84 -0.04 -1.66
```

However, if you only had c and e, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.loc["c":"e" + 1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design choice to make label-based slicing include both endpoints:

```
In [235]: s.loc["c":"e"]
Out[235]:
     c     d     e
  0 -0.84 -0.04 -1.66
```
This is most definitely a “practicality beats purity” sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

**Indexing potentially changes underlying Series dtype**

The different indexing operation can potentially change the dtype of a Series.

```python
In [236]: series1 = pd.Series([1, 2, 3])

In [237]: series1.dtype
Out[237]: dtype('int64')

In [238]: res = series1.reindex([0, 4])

In [239]: res.dtype
Out[239]: dtype('float64')

In [240]: res
Out[240]:
0    1.0
4    NaN
dtype: float64
```

```python
In [241]: series2 = pd.Series([True])

In [242]: series2.dtype
Out[242]: dtype('bool')

In [243]: res = series2.reindex_like(series1)

In [244]: res.dtype
Out[244]: dtype('O')

In [245]: res
Out[245]:
0   True
1   NaN
2   NaN
dtype: object
```

This is because the (re)indexing operations above silently inserts NaNs and the dtype changes accordingly. This can cause some issues when using numpy ufuncs such as `numpy.logical_and`.

See the GH2388 for a more detailed discussion.
2.7 Merge, join, concatenate and compare

pandas provides various facilities for easily combining together Series or DataFrame with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

In addition, pandas also provides utilities to compare two Series or DataFrame and summarize their differences.

2.7.1 Concatenating objects

The `concat()` function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say “if any” because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```python
In [1]: df1 = pd.DataFrame(
    ...:     {
    ...:         "B": ["B0", "B1", "B2", "B3"],
    ...:         "C": ["C0", "C1", "C2", "C3"],
    ...:         "D": ["D0", "D1", "D2", "D3"],
    ...:     },
    ...:     index=[0, 1, 2, 3],
    ...: )
In [2]: df2 = pd.DataFrame(
    ...:     {
    ...:         "A": ["A4", "A5", "A6", "A7"],
    ...:         "B": ["B4", "B5", "B6", "B7"],
    ...:         "C": ["C4", "C5", "C6", "C7"],
    ...:         "D": ["D4", "D5", "D6", "D7"],
    ...:     },
    ...:     index=[4, 5, 6, 7],
    ...: )
In [3]: df3 = pd.DataFrame(
    ...:     {
    ...:         "B": ["B8", "B9", "B10", "B11"],
    ...:         "C": ["C8", "C9", "C10", "C11"],
    ...:         "D": ["D8", "D9", "D10", "D11"],
    ...:     },
    ...:     index=[8, 9, 10, 11],
    ...: )
In [4]: frames = [df1, df2, df3]
In [5]: result = pd.concat(frames)
```
Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of “what to do with the other axes”:

```python
pd.concat(
    objs,
    axis=0,
    join="outer",
    ignore_index=False,
    keys=None,
    levels=None,
    names=None,
    verify_integrity=False,
    copy=True,
)
```

- **objs**: a sequence or mapping of Series or DataFrame objects. If a dict is passed, the sorted keys will be used as the `keys` argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.

- **axis**: `{0, 1, ...}`, default 0. The axis to concatenate along.

- **join**: {'inner', 'outer'}, default ‘outer’. How to handle indexes on other axis(es). Outer for union and inner for intersection.

- **ignore_index**: boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.
• **keys**: sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.

• **levels**: list of sequences, default None. Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.

• **names**: list, default None. Names for the levels in the resulting hierarchical index.

• **verify_integrity**: boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.

• **copy**: boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments don’t make much sense. Let’s revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the keys argument:

```python
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

As you can see (if you’ve read the rest of the documentation), the resulting object’s index has a **hierarchical index**. This means that we can now select out each chunk by key:

```python
In [7]: result.loc["y"]
```

**Out[7]:**

```
   A  B  C  D
4  A4  B4  C4  D4  
5  A5  B5  C5  D5  
6  A6  B6  C6  D6  
7  A7  B7  C7  D7  
```

As you can see (if you’ve read the rest of the documentation), the resulting object’s index has a **hierarchical index**. This means that we can now select out each chunk by key:
It's not a stretch to see how this can be very useful. More detail on this functionality below.

**Note:** It is worth noting that `concat()` (and therefore `append()`) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```python
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

**Note:** When concatenating DataFrames with named axes, pandas will attempt to preserve these index/column names whenever possible. In the case where all inputs share a common name, this name will be assigned to the result. When the input names do not all agree, the result will be unnamed. The same is true for `MultiIndex`, but the logic is applied separately on a level-by-level basis.

### Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following two ways:

- Take the union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.

Here is an example of each of these methods. First, the default `join='outer'` behavior:

```python
In [8]: df4 = pd.DataFrame(
    ...:     {
    ...:         "B": ["B2", "B3", "B6", "B7"],
    ...:         "D": ["D2", "D3", "D6", "D7"],
    ...:         "F": ["F2", "F3", "F6", "F7"],
    ...:     },
    ...:     index=[2, 3, 6, 7],
    ...: )
...
In [9]: result = pd.concat([df1, df4], axis=1)
```

<table>
<thead>
<tr>
<th>df1</th>
<th>df4</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>0</td>
<td>A0</td>
</tr>
<tr>
<td>A1</td>
<td>2</td>
<td>A1</td>
</tr>
<tr>
<td>A2</td>
<td>6</td>
<td>A2</td>
</tr>
<tr>
<td>A3</td>
<td>7</td>
<td>A3</td>
</tr>
<tr>
<td>NaN</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>B0</td>
<td>2</td>
<td>B0</td>
</tr>
<tr>
<td>B1</td>
<td>3</td>
<td>B1</td>
</tr>
<tr>
<td>B2</td>
<td>6</td>
<td>B2</td>
</tr>
<tr>
<td>B3</td>
<td>7</td>
<td>B3</td>
</tr>
<tr>
<td>NaN</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>C0</td>
<td>2</td>
<td>C0</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
<td>6</td>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
<td>7</td>
<td>C3</td>
</tr>
<tr>
<td>NaN</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>D0</td>
<td>2</td>
<td>D0</td>
</tr>
<tr>
<td>D1</td>
<td>3</td>
<td>D1</td>
</tr>
<tr>
<td>D2</td>
<td>6</td>
<td>D2</td>
</tr>
<tr>
<td>D3</td>
<td>7</td>
<td>D3</td>
</tr>
<tr>
<td>NaN</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>2</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>3</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>4</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>5</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Here is the same thing with `join='inner'`:
Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:

```python
In [11]: result = pd.concat([df1, df4], axis=1, join="inner")
```

Similarly, we could index before the concatenation:

```python
In [12]: pd.concat([df1, df4.reindex(df1.index)], axis=1)
```

```
Out[12]:
      A  B  C  D  B  D  F
0  A0 B0 C0 D0 NaN NaN
1  A1 B1 C1 D1 NaN NaN
2  A2 B2 C2 D2 B2 D2 F2
3  A3 B3 C3 D3 B3 D3 F3
```

Ignoring indexes on the concatenation axis

For DataFrame objects which don’t have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the `ignore_index` argument:

```python
In [13]: result = pd.concat([df1, df4], ignore_index=True, sort=False)
```

```
   A  B  C  D  B  D  F
0  A0 B0 C0 D0 NaN NaN
1  A1 B1 C1 D1 NaN NaN
2  A2 B2 C2 D2 B2 D2 F2
3  A3 B3 C3 D3 B3 D3 F3
```
Concatenating with mixed ndims

You can concatenate a mix of Series and DataFrame objects. The Series will be transformed to DataFrame with the column name as the name of the Series.

```python
In [14]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')
In [15]: result = pd.concat([df1, s1], axis=1)
```

**Note:** Since we're concatenating a Series to a DataFrame, we could have achieved the same result with DataFrame.assign(). To concatenate an arbitrary number of pandas objects (DataFrame or Series), use concat.

If unnamed Series are passed they will be numbered consecutively.

```python
In [16]: s2 = pd.Series(['_0', '_1', '_2', '_3'])
In [17]: result = pd.concat([df1, s2, s2, s2], axis=1)
```
Passing `ignore_index=True` will drop all name references.

```python
In [18]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```

More concatenating with group keys

A fairly common use of the `keys` argument is to override the column names when creating a new DataFrame based on existing Series. Notice how the default behaviour consists on letting the resulting DataFrame inherit the parent Series' name, when these existed.

```python
In [19]: s3 = pd.Series([0, 1, 2, 3], name="foo")
In [20]: s4 = pd.Series([0, 1, 2, 3])
In [21]: s5 = pd.Series([0, 1, 4, 5])
In [22]: pd.concat([s3, s4, s5], axis=1)
Out[22]:
foo 0 1 0 1 2 3 4
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
2 2 2 4 2 2 2 2
3 3 3 5 3 3 3 3
```

Through the `keys` argument we can override the existing column names.

```python
In [23]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
Out[23]:
red  blue  yellow
0    0    0    0
1    1    1    1
2    2    2    4
3    3    3    5
```
Let's consider a variation of the very first example presented:

\begin{verbatim}
In [24]: result = pd.concat(frames, keys=["x", "y", "z"])
\end{verbatim}

You can also pass a dict to `concat` in which case the dict keys will be used for the `keys` argument (unless other keys are specified):

\begin{verbatim}
In [25]: pieces = {"x": df1, "y": df2, "z": df3}

In [26]: result = pd.concat(pieces)
\end{verbatim}
In [27]: `result = pd.concat(pieces, keys=["z", "y"])

The MultiIndex created has levels that are constructed from the passed keys and the index of the DataFrame pieces:
In [28]: result.index.levels
Out[28]: FrozenList([['z', 'y'], [4, 5, 6, 7, 8, 9, 10, 11]])

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

In [29]: result = pd.concat(
    ....:     pieces, keys=['x', 'y', 'z'], levels=[['z', 'y', 'x', 'w']], names=['group_key'])

In [30]: result.index.levels
Out[30]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])

This is fairly esoteric, but it is actually necessary for implementing things like `GroupBy` where the order of a categorical variable is meaningful.
Appending rows to a DataFrame

If you have a series that you want to append as a single row to a DataFrame, you can convert the row into a DataFrame and use `concat`.

```python
In [31]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])
In [32]: result = pd.concat([df1, s2.to_frame().T], ignore_index=True)
```

You should use `ignore_index` with this method to instruct DataFrame to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed DataFrame and append or concatenate those objects.

2.7.2 Database-style DataFrame or named Series joining/merging

pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R). The reason for this is careful algorithmic design and the internal layout of the data in `DataFrame`.

See the cookbook for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a comparison with SQL.

pandas provides a single function, `merge()`, as the entry point for all standard database join operations between `DataFrame` or named `Series` objects:

```python
df.merge(
    left,
    right,
    how='inner',
    on=None,
    left_on=None,
    right_on=None,
    left_index=False,
    right_index=False,
    sort=True,
```

(continues on next page)
• **left**: A DataFrame or named Series object.

• **right**: Another DataFrame or named Series object.

• **on**: Column or index level names to join on. Must be found in both the left and right DataFrame and/or Series objects. If not passed and `left_index` and `right_index` are False, the intersection of the columns in the DataFrames and/or Series will be inferred to be the join keys.

• **left_on**: Columns or index levels from the left DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.

• **right_on**: Columns or index levels from the right DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.

• **left_index**: If True, use the index (row labels) from the left DataFrame or Series as its join key(s). In the case of a DataFrame or Series with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame or Series.

• **right_index**: Same usage as `left_index` for the right DataFrame or Series

• **how**: One of 'left', 'right', 'outer', 'inner', 'cross'. Defaults to inner. See below for more detailed description of each method.

• **sort**: Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve performance substantially in many cases.

• **suffixes**: A tuple of string suffixes to apply to overlapping columns. Defaults to ('_x', '_y').

• **copy**: Always copy data (default True) from the passed DataFrame or named Series objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.

• **indicator**: Add a column to the output DataFrame called `_merge` with information on the source of each row. `_merge` is Categorical-type and takes on a value of `left_only` for observations whose merge key only appears in 'left' DataFrame or Series, `right_only` for observations whose merge key only appears in 'right' DataFrame or Series, and `both` if the observation’s merge key is found in both.

• **validate**: string, default None. If specified, checks if merge is of specified type.
  - “one_to_one” or “1:1”: checks if merge keys are unique in both left and right datasets.
  - “one_to_many” or “1:m”: checks if merge keys are unique in left dataset.
  - “many_to_one” or “m:1”: checks if merge keys are unique in right dataset.
  - “many_to_many” or “m:m”: allowed, but does not result in checks.

**Note**: Support for specifying index levels as the on, left_on, and right_on parameters was added in version 0.23.0. Support for merging named Series objects was added in version 0.24.0.

The return type will be the same as `left`. If `left` is a DataFrame or named Series and `right` is a subclass of DataFrame, the return type will still be DataFrame.
merge is a function in the pandas namespace, and it is also available as a DataFrame instance method `merge()`, with the calling DataFrame being implicitly considered the left object in the join.

The related `join()` method, uses `merge` internally for the index-on-index (by default) and column(s)-on-index join. If you are joining on index only, you may wish to use DataFrame.join to save yourself some typing.

**Brief primer on merge methods (relational algebra)**

Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (DataFrame objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two DataFrame objects on their indexes (which must contain unique values).
- **many-to-one** joins: for example when joining an index (unique) to one or more columns in a different DataFrame.
- **many-to-many** joins: joining columns on columns.

**Note:** When joining columns on columns (potentially a many-to-many join), any indexes on the passed DataFrame objects will be discarded.

It is worth spending some time understanding the result of the many-to-many join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the Cartesian product of the associated data. Here is a very basic example with one unique key combination:

```python
In [33]: left = pd.DataFrame(
    ...:     {
    ...:         "key": ["K0", "K1", "K2", "K3"],
    ...:         "B": ["B0", "B1", "B2", "B3"],
    ...:     }
    ...: )
    ...:

In [34]: right = pd.DataFrame(
    ...:     {
    ...:         "key": ["K0", "K1", "K2", "K3"],
    ...:         "C": ["C0", "C1", "C2", "C3"],
    ...:         "D": ["D0", "D1", "D2", "D3"],
    ...:     }
    ...: )
    ...:

In [35]: result = pd.merge(left, right, on="key")
```
Here is a more complicated example with multiple join keys. Only the keys appearing in `left` and `right` are present (the intersection), since `how='inner'` by default.

```python
In [36]: left = pd.DataFrame(
    ....:     {  
    ....:         "key1": ["K0", "K0", "K1", "K2"],  
    ....:         "key2": ["K0", "K1", "K0", "K1"],  
    ....:         "B": ["B0", "B1", "B2", "B3"],  
    ....:     })
    ....: 

In [37]: right = pd.DataFrame(
    ....:     {  
    ....:         "key1": ["K0", "K1", "K1", "K2"],  
    ....:         "key2": ["K0", "K0", "K0", "K0"],  
    ....:         "C": ["C0", "C1", "C2", "C3"],  
    ....:         "D": ["D0", "D1", "D2", "D3"],  
    ....:     })
    ....: 

In [38]: result = pd.merge(left, right, on=['"key1"', '"key2"'])
```

The `how` argument to `merge` specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be `NA`. Here is a summary of the `how` options and their SQL equivalent names:

<table>
<thead>
<tr>
<th><code>how</code></th>
<th>SQL Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inner</code></td>
<td>INNER</td>
</tr>
<tr>
<td><code>left</code></td>
<td>LEFT</td>
</tr>
<tr>
<td><code>right</code></td>
<td>RIGHT</td>
</tr>
<tr>
<td><code>outer</code></td>
<td>OUTER</td>
</tr>
<tr>
<td><code>cross</code></td>
<td>CROSS</td>
</tr>
</tbody>
</table>

```
### Merge method

<table>
<thead>
<tr>
<th>Method</th>
<th>SQL Join Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>LEFT OUTER JOIN</td>
<td>Use keys from left frame only</td>
</tr>
<tr>
<td>right</td>
<td>RIGHT OUTER JOIN</td>
<td>Use keys from right frame only</td>
</tr>
<tr>
<td>outer</td>
<td>FULL OUTER JOIN</td>
<td>Use union of keys from both frames</td>
</tr>
<tr>
<td>inner</td>
<td>INNER JOIN</td>
<td>Use intersection of keys from both frames</td>
</tr>
<tr>
<td>cross</td>
<td>CROSS JOIN</td>
<td>Create the cartesian product of rows of both frames</td>
</tr>
</tbody>
</table>

#### In [39]: result = pd.merge(left, right, how="left", on=["key1", "key2"])

#### In [40]: result = pd.merge(left, right, how="right", on=["key1", "key2"])

#### In [41]: result = pd.merge(left, right, how="outer", on=["key1", "key2"])

---

522 Chapter 2. User Guide
In [42]: result = pd.merge(left, right, how="inner", on=['key1', 'key2'])

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>K0</td>
<td>A0</td>
<td>B0</td>
</tr>
<tr>
<td>1</td>
<td>K1</td>
<td>A1</td>
<td>B1</td>
</tr>
<tr>
<td>2</td>
<td>K2</td>
<td>A2</td>
<td>B2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>K0</td>
<td>C0</td>
<td>D0</td>
</tr>
<tr>
<td>1</td>
<td>K1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>2</td>
<td>K2</td>
<td>C2</td>
<td>D2</td>
</tr>
</tbody>
</table>

You can merge a multi-indexed Series and a DataFrame, if the names of the MultiIndex correspond to the columns from the DataFrame. Transform the Series to a DataFrame using `Series.reset_index()` before merging, as shown in the following example.

In [44]: df = pd.DataFrame({"Let": ["A", "B", "C"], "Num": [1, 2, 3]})

In [45]: df
Out[45]:
<table>
<thead>
<tr>
<th>Let</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>
In [46]: ser = pd.Series(
    ....:     ["a", "b", "c", "d", "e", "f"],
    ....:     index=pd.MultiIndex.from_arrays(
    ....:         ["A", "B", "C"] * 2, [1, 2, 3, 4, 5, 6]), names=["Let", "Num"]
    ....: ),
    ....: )

In [47]: ser
Out[47]:
Let  Num
A   1   a
B   2   b
C   3   c
A   4   d
B   5   e
C   6   f
dtype: object

In [48]: pd.merge(df, ser.reset_index(), on=["Let", "Num"])
Out[48]:
   Let  Num  0
0   A    1   a
1   B    2   b
2   C    3   c

Here is another example with duplicate join keys in DataFrames:

In [49]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})

In [50]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})

In [51]: result = pd.merge(left, right, on="B", how="outer")

left  right  Result
0  1  2  4
1  4  3
2  5  2
3  6  2
0  1  2  5
1  1  2  6
2  2  2  4
3  2  2  5
4  2  2  6
**Warning:** Joining / merging on duplicate keys can cause a returned frame that is the multiplication of the row dimensions, which may result in memory overflow. It is the user’s responsibility to manage duplicate values in keys before joining large DataFrames.

### Checking for duplicate keys

Users can use the `validate` argument to automatically check whether there are unexpected duplicates in their merge keys. Key uniqueness is checked before merge operations and so should protect against memory overflows. Checking key uniqueness is also a good way to ensure user data structures are as expected.

In the following example, there are duplicate values of B in the right DataFrame. As this is not a one-to-one merge – as specified in the `validate` argument – an exception will be raised.

```python
In [52]: left = pd.DataFrame({"A": [1, 2], "B": [1, 2]})
In [53]: right = pd.DataFrame({"A": [4, 5, 6], "B": [2, 2, 2]})
In [53]: result = pd.merge(left, right, on="B", how="outer", validate="one_to_one")
...:
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right DataFrame but wants to ensure there are no duplicates in the left DataFrame, one can use the `validate='one_to_many'` argument instead, which will not raise an exception.

```python
In [54]: pd.merge(left, right, on="B", how="outer", validate="one_to_many")
Out[54]:
   A_x  B  A_y
0   1   1   NaN
1   2   2   4.0
2   2   2   5.0
3   2   2   6.0
```

### The merge indicator

`merge()` accepts the argument `indicator`. If True, a Categorical-type column called `_merge` will be added to the output object that takes on values:

<table>
<thead>
<tr>
<th>Observation Origin</th>
<th>_merge value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge key only in 'left' frame</td>
<td>left_only</td>
</tr>
<tr>
<td>Merge key only in 'right' frame</td>
<td>right_only</td>
</tr>
<tr>
<td>Merge key in both frames</td>
<td>both</td>
</tr>
</tbody>
</table>

```python
In [55]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})
In [56]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})
In [57]: pd.merge(df1, df2, on="col1", how="outer", indicator=True)
Out[57]:
   col1  col_left  col_right  _merge
0    0        a          2  left_only
1    1        b          2  right_only
2    1        a          2    both
3    2        b          2    both
```

(continues on next page)
The indicator argument will also accept string arguments, in which case the indicator function will use the value of the passed string as the name for the indicator column.

In [58]: pd.merge(df1, df2, on="col1", how="outer", indicator="indicator_column")

Out[58]:
<table>
<thead>
<tr>
<th>col1</th>
<th>col_left</th>
<th>col_right</th>
<th>indicator_column</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>NaN</td>
<td>left_only</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>2.0</td>
<td>both</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>2.0</td>
<td>right_only</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>2.0</td>
<td>right_only</td>
</tr>
</tbody>
</table>

**Merge dtypes**

Merging will preserve the dtype of the join keys.

In [59]: left = pd.DataFrame({"key": [1], "v1": [10]})

In [60]: left
Out[60]:
<table>
<thead>
<tr>
<th>key</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

In [61]: right = pd.DataFrame({"key": [1, 2], "v1": [20, 30]})

In [62]: right
Out[62]:
<table>
<thead>
<tr>
<th>key</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

We are able to preserve the join keys:

In [63]: pd.merge(left, right, how="outer")
Out[63]:
<table>
<thead>
<tr>
<th>key</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
</tbody>
</table>

In [64]: pd.merge(left, right, how="outer").dtypes
Out[64]:
<table>
<thead>
<tr>
<th>key</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>int64</td>
</tr>
<tr>
<td>dtype: object</td>
<td></td>
</tr>
</tbody>
</table>

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.
Merging will preserve category dtypes of the mergands. See also the section on `categoricals`.

The left frame.

```python
In [67]: from pandas.api.types import CategoricalDtype

In [68]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [69]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))

In [70]: left = pd.DataFrame(
            ....:     {"X": X, "Y": np.random.choice(['one', 'two', 'three'], size=(10,))})

In [71]: left
Out[71]:
   X     Y
0  bar  one
1   foo  one
2  foo  three
3   bar  three
4  foo  one
5   bar  one
6  bar  three
7   bar  three
8  bar  three
9   foo  three

In [72]: left.dtypes
Out[72]:
   X         category
   Y            object
dtype: object
```

The right frame.

```python
In [73]: right = pd.DataFrame(
            ....:     {
            ....:         "X": pd.Series(['foo', 'bar'], dtype=CategoricalDtype(['foo', 'bar'])),
            ....:         "Z": [1, 2],
            ....:     })
```
In [74]: right
Out[74]:
   X  Z
0  foo  1
1  bar  2

In [75]: right.dtypes
Out[75]:
   X  category
   Z  int64
dtype: object

The merged result:

In [76]: result = pd.merge(left, right, how="outer")

In [77]: result
Out[77]:
   X  Y  Z
0  bar  one  2
1  bar  three  2
2  bar  one  2
3  bar  two  2
4  bar  three  2
5  bar  three  2
6  foo  one  1
7  foo  three  1
8  foo  one  1
9  foo  three  1

In [78]: result.dtypes
Out[78]:
   X  category
   Y  object
   Z  int64
dtype: object

**Note:** The category dtypes must be exactly the same, meaning the same categories and the ordered attribute. Otherwise the result will coerce to the categories’ dtype.

**Note:** Merging on category dtypes that are the same can be quite performant compared to object dtype merging.
Joining on index

`DataFrame.join()` is a convenient method for combining the columns of two potentially differently-indexed `DataFrames` into a single result `DataFrame`. Here is a very basic example:

```python
In [79]: left = pd.DataFrame(
    ....:    {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"],
    ....:     index=["K0", "K1", "K2"]
    ....: )

In [80]: right = pd.DataFrame(
    ....:    {"C": ["C0", "C2", "C3"], "D": ["D0", "D2", "D3"],
    ....:     index=["K0", "K2", "K3"]
    ....: )

In [81]: result = left.join(right)
```

The data alignment here is on the indexes (row labels). This same behavior can be achieved using `merge` plus additional arguments instructing it to use the indexes:
In [84]: result = pd.merge(left, right, left_index=True, right_index=True, how="outer")

Joining key columns on an index

`join()` takes an optional on argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame. These two function calls are completely equivalent:

```python
left.join(right, on=key_or_keys)
pd.merge(
    left, right, left_on=key_or_keys, right_index=True, how="left", sort=False
)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the DataFrame’s is already indexed by the join key), using join may be more convenient. Here is a simple example:

In [86]: left = pd.DataFrame(
    ...:     {
    ...:         "B": ["B0", "B1", "B2", "B3"],
    ...:         "key": ["K0", "K1", "K0", "K1"],
    ...:     }
    ...: )

In [87]: right = pd.DataFrame({"C": ["C0", "C1"], "D": ["D0", "D1"], index=["K0", "K1"]})

In [88]: result = left.join(right, on="key")
In [89]: result = pd.merge(
    ....:     left, right, left_on="key", right_index=True, how="left", sort=False
    ....: )

In [90]: left = pd.DataFrame(
    ....:     {
    ....:         "B": ["B0", "B1", "B2", "B3"],
    ....:         "key1": ["K0", "K0", "K1", "K2"],
    ....:         "key2": ["K0", "K1", "K0", "K1"],
    ....:     }
    ....: )

In [91]: index = pd.MultiIndex.from_tuples(
    ....:     ["K0", "K0"], ["K1", "K0"], ["K2", "K0"], ["K2", "K1"]
    ....: )

In [92]: right = pd.DataFrame(
    ....:     {
    ....:         "C": ["C0", "C1", "C2", "C3"],
    ....:         "D": ["D0", "D1", "D2", "D3"],
    ....:     }, index=index
    ....: )

Now this can be joined by passing the two key column names:

In [93]: result = left.join(right, on=["key1", "key2"])
The default for DataFrame.join is to perform a left join (essentially a “VLOOKUP” operation, for Excel users), which uses only the keys found in the calling DataFrame. Other join types, for example inner join, can be just as easily performed:

```
In [94]: result = left.join(right, on=['key1', 'key2'], how='inner')
```

As you can see, this drops any rows where there was no match.

**Joining a single Index to a MultiIndex**

You can join a singly-indexed DataFrame with a level of a MultiIndexed DataFrame. The level will match on the name of the index of the singly-indexed frame against a level name of the MultiIndexed frame.

```
In [95]: left = pd.DataFrame(
     ....:     columns=['A', 'B', 'key1', 'key2'],
     ....:     index=[0, 1, 2, 3],
     ....:     data=[('A0', 'B0', 'K0', 'K0'), ('A1', 'B1', 'K0', 'K1'), ('A2', 'B2', 'K1', 'K0'), ('A3', 'B3', 'K2', 'K1')],
     ....:     index_names=['key'],
     ....:     )

In [96]: index = pd.MultiIndex.from_tuples(
     ....:     [(0, 'Y0'), (1, 'Y1'), (2, 'Y2'), (3, 'Y3')],
     ....:     names=['key', 'Y'],
     ....:     )

In [97]: right = pd.DataFrame(
     ....:     columns=['C', 'D'],
     ....:     index=index,
     ....:     )

In [98]: result = left.join(right, how='inner')
```
This is equivalent but less verbose and more memory efficient / faster than this.

```python
In [99]: result = pd.merge(
       ....:     left.reset_index(), right.reset_index(), on="key", how="inner"
       ....:     ).set_index(["key", "Y"])
       ....:
```

### Joining with two MultiIndexes

This is supported in a limited way, provided that the index for the right argument is completely used in the join, and is a subset of the indices in the left argument, as in this example:

```python
In [100]: leftindex = pd.MultiIndex.from_product(
       ....:     [list("abc"), list("xy"), [1, 2]], names=["abc", "xy", "num"]
       ....:     )
       ....:

In [101]: left = pd.DataFrame({"v1": range(12)}, index=leftindex)

In [102]: left
Out[102]:
          v1
    abc  xy  num
   a  x  1  0
    2  1
   y  1  2
    2  3
   b  x  4  0
    2  5
   y  1  6
    2  7
```

(continues on next page)
In [103]: rightindex = pd.MultiIndex.from_product(  
.....:     [list("abc"), list("xy")], names=["abc", "xy"]  
.....: )
.....:

In [104]: right = pd.DataFrame({"v2": [100 * i for i in range(1, 7)]}, index=rightindex)

In [105]: right
Out[105]:
      v2
abc xy
a  x 100
  y 200
b  x 300
  y 400
c  x 500
  y 600

In [106]: left.join(right, on=["abc", "xy"], how="inner")
Out[106]:
      v1  v2
abc xy num
a  x 1 0 100
  2 1 100
  y 1 2 200
  2 3 200
b  x 1 4 300
  2 5 300
  y 1 6 400
  2 7 400
c  x 1 8 500
  2 9 500
  y 1 10 600
  2 11 600

If that condition is not satisfied, a join with two multi-indexes can be done using the following code.

In [107]: leftindex = pd.MultiIndex.from_tuples(  
.....:     [("K0", "X0"), ("K0", "X1"), ("K1", "X2")], names=["key", "X"]
.....: )
.....:

In [108]: left = pd.DataFrame(  
.....:     {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"]}, index=leftindex  
.....: )
.....:

(continues on next page)
Merging on a combination of columns and index levels

Strings passed as the on, left_on, and right_on parameters may refer to either column names or index level names. This enables merging DataFrame instances on a combination of index levels and columns without resetting indexes.

```
In [110]: left_index = pd.Index(["K0", "K1", "K2"], name="key1")
In [111]: left = pd.DataFrame(
    ....:     {"A": ["A0", "A1", "A2", "A3"],
    ....:      "B": ["B0", "B1", "B2", "B3"],
    ....:      "key2": ["K0", "K1", "K0", "K1"],
    ....:     },
    ....:     index=left_index,
    ....: )
In [112]: right_index = pd.Index(["K0", "K1", "K2"], name="key1")
In [113]: right = pd.DataFrame(
    ....:     {"C": ["C0", "C1", "C2", "C3"],
    ....:      "D": ["D0", "D1", "D2", "D3"],
    ....:     },
    ....:     index=right_index,
    ....: )
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [116]: result = left.merge(right, on="key1", key2")

Note: When DataFrames are merged on a string that matches an index level in both frames, the index level is preserved as an index level in the resulting DataFrame.

Note: When DataFrames are merged using only some of the levels of a MultiIndex, the extra levels will be dropped from the resulting merge. In order to preserve those levels, use reset_index on those level names to move those levels to columns prior to doing the merge.

Note: If a string matches both a column name and an index level name, then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

Overlapping value columns

The merge suffixes argument takes a tuple of list of strings to append to overlapping column names in the input DataFrames to disambiguate the result columns:

In [117]: left = pd.DataFrame({"k": ["K0", "K1", "K2"], "v": [1, 2, 3]})

In [118]: right = pd.DataFrame({"k": ["K0", "K0", "K3"], "v": [4, 5, 6]})

In [119]: result = pd.merge(left, right, on="k")
Joining multiple DataFrames

A list or tuple of DataFrames can also be passed to `join()` to join them together on their indexes.

In [124]: right2 = pd.DataFrame({"v": [7, 8, 9]}, index=["K1", "K1", "K2"])

In [125]: result = left.join([right, right2])
Merging together values within Series or DataFrame columns

Another fairly common situation is to have two like-indexed (or similarly indexed) Series or DataFrame objects and wanting to “patch” values in one object from values for matching indices in the other. Here is an example:

```python
In [126]: df1 = pd.DataFrame(  
.....:     [[np.nan, 3.0, 5.0], [-4.6, np.nan, np.nan], [np.nan, 7.0, np.nan]]  
.....: )
.....:
In [127]: df2 = pd.DataFrame([-42.6, np.nan, -8.2], [-5.0, 1.6, 4], index=[1, 2])
```

For this, use the `combine_first()` method:

```python
In [128]: result = df1.combine_first(df2)
```

Note that this method only takes values from the right DataFrame if they are missing in the left DataFrame. A related method, `update()`, alters non-NA values in place:

```python
In [129]: df1.update(df2)
```
2.7.3 Timeseries friendly merging

Merging ordered data

A `merge_ordered()` function allows combining time series and other ordered data. In particular it has an optional `fill_method` keyword to fill/interpolate missing data:

```
In [130]: left = pd.DataFrame(
    .....:   {"k": ["K0", "K1", "K1", "K2"], "lv": [1, 2, 3, 4], "s": ["a", "b", "c", "d"]}
    .....: )
    .....:
In [131]: right = pd.DataFrame({"k": ["K1", "K2", "K4"], "rv": [1, 2, 3]})

In [132]: pd.merge_ordered(left, right, fill_method="ffill", left_by="s")
Out[132]:
    k  lv  s  rv
0  K0  1.0  a  NaN
1  K1  1.0  a  1.0
2  K2  1.0  a  2.0
3  K4  1.0  a  3.0
4  K1  2.0  b  1.0
5  K2  2.0  b  2.0
6  K4  2.0  b  3.0
7  K1  3.0  c  1.0
8  K2  3.0  c  2.0
9  K4  3.0  c  3.0
10 K1  NaN  d  1.0
11 K2  4.0  d  2.0
12 K4  4.0  d  3.0
```

Merging asof

A `merge_asof()` is similar to an ordered left-join except that we match on nearest key rather than equal keys. For each row in the `left DataFrame`, we select the last row in the `right DataFrame` whose `on key` is less than the left's key. Both DataFrames must be sorted by the key.

Optionally an asof merge can perform a group-wise merge. This matches the by key equally, in addition to the nearest match on the on key.

For example; we might have trades and quotes and we want to asof merge them.

```
In [133]: trades = pd.DataFrame(
    .....:   {"time": pd.to_datetime(
    .....:       "20160525 13:30:00.023",
    .....:       "20160525 13:30:00.038",
    .....:       "20160525 13:30:00.048",
    .....:       "20160525 13:30:00.048",
    .....:       "20160525 13:30:00.048",
    .....:       "20160525 13:30:00.048",
    .....:   })
    .....:
```

(continues on next page)
In [134]: quotes = pd.DataFrame优点
In [135]: trades
Out[135]:
<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.038</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.048</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
</tr>
</tbody>
</table>
In [136]: quotes
Out[136]:
<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-05-25 13:30:00.023</td>
<td>GOOG</td>
<td>720.50</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>51.96</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.030</td>
<td>MSFT</td>
<td>51.97</td>
<td>51.98</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.041</td>
<td>MSFT</td>
<td>51.99</td>
<td>52.00</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.93</td>
<td>720.93</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.049</td>
<td>AAPL</td>
<td>97.99</td>
<td>98.01</td>
</tr>
<tr>
<td>2016-05-25 13:30:00.072</td>
<td>GOOG</td>
<td>720.50</td>
<td>720.88</td>
</tr>
</tbody>
</table>
By default we are taking the asof of the quotes.

```python
In [137]: pd.merge_asof(trades, quotes, on="time", by="ticker")
Out[137]:
                  time  ticker  price  quantity  bid  ask
0  2016-05-25 13:30:00.023  MSFT  51.95      75  51.95  51.96
1  2016-05-25 13:30:00.038  MSFT  51.95     155  51.97  51.98
2  2016-05-25 13:30:00.048  GOOG  720.77     100  720.50  720.93
3  2016-05-25 13:30:00.048  GOOG  720.92     100  720.50  720.93
4  2016-05-25 13:30:00.048  AAPL   98.00     100  NaN   NaN
```

We only asof within 2ms between the quote time and the trade time.

```python
In [138]: pd.merge_asof(trades, quotes, on="time", by="ticker", tolerance=pd.Timedelta("2ms"))
Out[138]:
                  time  ticker  price  quantity  bid  ask
0  2016-05-25 13:30:00.023  MSFT  51.95      75  51.95  51.96
1  2016-05-25 13:30:00.038  MSFT  51.95     155  NaN   NaN
2  2016-05-25 13:30:00.048  GOOG  720.77     100  720.50  720.93
3  2016-05-25 13:30:00.048  GOOG  720.92     100  720.50  720.93
4  2016-05-25 13:30:00.048  AAPL   98.00     100  NaN   NaN
```

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. Note that though we exclude the exact matches (of the quotes), prior quotes do propagate to that point in time.

```python
In [139]: pd.merge_asof(        trades,        quotes,        on="time",        by="ticker",        tolerance=pd.Timedelta("10ms"),        allow_exact_matches=False,    )
Out[139]:
                  time  ticker  price  quantity  bid  ask
0  2016-05-25 13:30:00.023  MSFT  51.95      75  NaN   NaN
1  2016-05-25 13:30:00.038  MSFT  51.95     155  51.97  51.98
2  2016-05-25 13:30:00.048  GOOG  720.77     100  720.50  720.93
3  2016-05-25 13:30:00.048  GOOG  720.92     100  720.50  720.93
4  2016-05-25 13:30:00.048  AAPL   98.00     100  NaN   NaN
```

### 2.7.4 Comparing objects

The `compare()` and `compare()` methods allow you to compare two DataFrame or Series, respectively, and summarize their differences.

This feature was added in **V1.1.0**.

For example, you might want to compare two `DataFrame` and stack their differences side by side.

```python
In [140]: df = pd.DataFrame(
        {
(continues on next page)
In [141]: df
Out[141]:
    col1 col2 col3
0     a  1.0  1.0
1     a  2.0  2.0
2     b  3.0  3.0
3     b  NaN  4.0
4     a  5.0  5.0

In [142]: df2 = df.copy()

In [143]: df2.loc[0, "col1"] = "c"

In [144]: df2.loc[2, "col3"] = 4.0

In [145]: df2
Out[145]:
    col1 col2 col3
0     c  1.0  1.0
1     a  2.0  2.0
2     b  3.0  4.0
3     b  NaN  4.0
4     a  5.0  5.0

In [146]: df.compare(df2)
Out[146]:
    col1 col3
  self other self other
0     a     c    NaN NaN
2  NaN     NaN  3.0  4.0

By default, if two corresponding values are equal, they will be shown as NaN. Furthermore, if all values in an entire row / column, the row / column will be omitted from the result. The remaining differences will be aligned on columns.

If you wish, you may choose to stack the differences on rows.

In [147]: df.compare(df2, align_axis=0)
Out[147]:
    col1 col3
  self other self other
0     a NaN    NaN NaN
2  NaN NaN  3.0  4.0

If you wish to keep all original rows and columns, set keep_shape argument to True.
In [148]: df.compare(df2, keep_shape=True)
Out[148]:
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>self</td>
<td>other</td>
<td>self</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>c</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

You may also keep all the original values even if they are equal.

In [149]: df.compare(df2, keep_shape=True, keep_equal=True)
Out[149]:
<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>self</td>
<td>other</td>
<td>self</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>c</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>b</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>a</td>
<td>5.0</td>
</tr>
</tbody>
</table>

2.8 Reshaping and pivot tables

2.8.1 Reshaping by pivoting DataFrame objects

Pivot

```
df = df.pivot(index='foo', columns='bar', values='baz')
```

Data is often stored in so-called “stacked” or “record” format:
In [1]: import pandas._testing as tm

In [2]: def unpivot(frame):
   ...:     N, K = frame.shape
   ...:     data = {
   ...:         "value": frame.to_numpy().ravel("F"),
   ...:         "variable": np.asarray(frame.columns).repeat(N),
   ...:         "date": np.tile(np.asarray(frame.index), K),
   ...:     }
   ...:     return pd.DataFrame(data, columns=["date", "variable", "value"])

In [3]: df = unpivot(tm.makeTimeDataFrame(3))

In [4]: df
Out[4]:
   date    variable  value
0 2000-01-03       A  0.469112
1 2000-01-04       A -0.282863
2 2000-01-05       A -1.509059
3 2000-01-03       B -1.135632
4 2000-01-04       B  1.212112
5 2000-01-05       B -0.173215
6 2000-01-03       C  0.119209
7 2000-01-04       C -1.044236
8 2000-01-05       C -0.861849
9 2000-01-03       D -2.104569
10 2000-01-04      D -0.494929
11 2000-01-05      D  1.071804

To select out everything for variable A we could do:

In [5]: filtered = df[df["variable"] == "A"]

In [6]: filtered
Out[6]:
   date    variable  value
0 2000-01-03       A  0.469112
1 2000-01-04       A -0.282863
2 2000-01-05       A -1.509059

But suppose we wish to do time series operations with the variables. A better representation would be where the columns are the unique variables and an index of dates identifies individual observations. To reshape the data into this form, we use the DataFrame.pivot() method (also implemented as a top level function pivot()):

In [7]: pivoted = df.pivot(index="date", columns="variable", values="value")

In [8]: pivoted
Out[8]:
   variable  A   B   C    D
date
2000-01-03  0.469112 -1.135632 0.119209 -2.104569
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929

(continues on next page)
If the `values` argument is omitted, and the input `DataFrame` has more than one column of values which are not used as column or index inputs to `pivot()`, then the resulting “pivoted” `DataFrame` will have hierarchical columns whose topmost level indicates the respective value column:

```python
In [9]: df["value2"] = df["value"] * 2
In [10]: pivoted = df.pivot(index="date", columns="variable")
In [11]: pivoted
Out[11]:
   value   value2
variable  A    B    C    D  A    B    C
-------  ----  ----  ----  ----  ----  ----
      D date
  2000-01-03  0.469112 -1.135632 0.119209 -2.104569 0.938225 -2.271265 0.238417 -4.209138
  2000-01-04 -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224 -2.088472 -0.989859
  2000-01-05 -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429 -1.723698  2.143608
```

You can then select subsets from the pivoted `DataFrame`:

```python
In [12]: pivoted["value2"]
Out[12]:
    variable  A    B    C    D
date
  2000-01-03  0.469112 -1.135632 0.119209 -2.104569 0.938225 -2.271265 0.238417 -4.209138
  2000-01-04 -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224 -2.088472 -0.989859
  2000-01-05 -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429 -1.723698  2.143608
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

**Note:** `pivot()` will error with a `ValueError`: Index contains duplicate entries, cannot reshape if the index/column pair is not unique. In this case, consider using `pivot_table()` which is a generalization of pivot that can handle duplicate values for one index/column pair.
2.8.2 Reshaping by stacking and unstacking

Stack

Closely related to the \texttt{pivot()} method are the related \texttt{stack()} and \texttt{unstack()} methods available on \texttt{Series} and \texttt{DataFrame}. These methods are designed to work together with \texttt{MultiIndex} objects (see the section on \textit{hierarchical indexing}). Here are essentially what these methods do:

- **\texttt{stack()}:** “pivot” a level of the (possibly hierarchical) column labels, returning a \texttt{DataFrame} with an index with a new inner-most level of row labels.

- **\texttt{unstack()}:** (inverse operation of \texttt{stack()}) “pivot” a level of the (possibly hierarchical) row index to the column axis, producing a reshaped \texttt{DataFrame} with a new inner-most level of column labels.
The clearest way to explain is by example. Let’s take a prior example data set from the hierarchical indexing section:

```
In [13]: tuples = list(
    ....:     zip(
    ....:         ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
    ....:         ["one", "two", "one", "two", "one", "two", "one", "two"],
    ....:     )
    ....: )
    ....:

In [14]: index = pd.MultiIndex.from_tuples(tuples, names=["first", "second")

In [15]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])

In [16]: df2 = df[:4]

In [17]: df2
Out[17]:
       A         B
    first second
    bar  one  0.721555 -0.706771
    two  one  -1.039575  0.271860
    baz  one -0.424972  0.567020
    two  one  0.276232 -1.087401
```

The `stack()` function “compresses” a level in the DataFrame columns to produce either:

- A `Series`, in the case of a simple column Index.
- A DataFrame, in the case of a MultiIndex in the columns.

If the columns have a MultiIndex, you can choose which level to stack. The stacked level becomes the new lowest level in a MultiIndex on the columns:

```
in [18]: stacked = df2.stack()

In [19]: stacked
Out[19]:
first   second
bar   one  A  0.721555
       B -0.706771
       two A -1.039575
             B  0.271860
baz   one  A -0.424972
       B  0.567020
       two A  0.276232
             B -1.087401

```

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack() is unstack(), which by default unstacks the last level:

```
in [20]: stacked.unstack()
Out[20]:
first   second
bar   one  A  0.721555 -0.706771
       two -1.039575  0.271860
baz   one -0.424972  0.567020
       two  0.276232 -1.087401

In [21]: stacked.unstack(1)
Out[21]:
second   one   two
first
bar   A  0.721555 -1.039575
       B -0.706771  0.271860
baz   A -0.424972  0.276232
       B  0.567020 -1.087401

In [22]: stacked.unstack(0)
Out[22]:
first   bar   baz
second
one   A  0.721555 -0.424972
       B -0.706771  0.567020
two   A -1.039575  0.276232
       B  0.271860 -1.087401
```
If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [23]: stacked.unstack("second")
Out[23]:
   second  one    two
first
bar     A  0.721555 -1.039575
       B -0.706771  0.271860
baz     A -0.424972  0.276232
       B  0.567020 -1.087401
```
Unstack(θ)

Notice that the `stack()` and `unstack()` methods implicitly sort the index levels involved. Hence a call to `stack()` and then `unstack()`, or vice versa, will result in a sorted copy of the original `DataFrame` or `Series`:

```python
In [24]: index = pd.MultiIndex.from_product([[2, 1], ["a", "b"]])
In [25]: df = pd.DataFrame(np.random.randn(4), index=index, columns=["A"])
In [26]: df
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 a</td>
<td>-0.370647</td>
</tr>
<tr>
<td>2 b</td>
<td>-1.157892</td>
</tr>
<tr>
<td>1 a</td>
<td>-1.344312</td>
</tr>
<tr>
<td>1 b</td>
<td>0.844885</td>
</tr>
</tbody>
</table>

```python
Out[26]:
```

```python
In [27]: all(df.unstack().stack() == df.sort_index())
```

```
Out[27]: True
```

The above code will raise a `TypeError` if the call to `sort_index()` is removed.

**Multiple levels**

You may also stack or unstack more than one level at a time by passing a list of levels, in which case the end result is as if each level in the list were processed individually.

```python
In [28]: columns = pd.MultiIndex.from_tuples(
    ....:     ["A", "cat", "long"],
    ....:     ["B", "cat", "long"],
    ....:     ["A", "dog", "short"],
```
```python
In [29]: df = pd.DataFrame(np.random.randn(4, 4), columns=columns)
In [30]: df
Out[30]:
exp  A   B   A   B
animal  cat  cat  dog  dog
hair_length  long  long  short  short
0  1.075770 -0.109050  1.643563 -1.469388
1  0.357021 -0.674600 -1.776904 -0.968914
2 -1.294524  0.413738  0.276662 -0.472035
3 -0.013960 -0.362543 -0.006154 -0.923061
In [31]: df.stack(level=['animal', 'hair_length'])
Out[31]:
exp   A   B
animal hair_length
0      cat  long  1.075770 -0.109050
       dog  short  1.643563 -1.469388
1      cat  long  0.357021 -0.674600
       dog  short -1.776904 -0.968914
2      cat  long -1.294524  0.413738
       dog  short  0.276662 -0.472035
3      cat  long -0.013960 -0.362543
       dog  short -0.006154 -0.923061
```

The list of levels can contain either level names or level numbers (but not a mixture of the two).

```python
# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [32]: df.stack(level=[1, 2])
Out[32]:
exp   A   B
animal hair_length
0      cat  long  1.075770 -0.109050
       dog  short  1.643563 -1.469388
1      cat  long  0.357021 -0.674600
       dog  short -1.776904 -0.968914
2      cat  long -1.294524  0.413738
       dog  short  0.276662 -0.472035
3      cat  long -0.013960 -0.362543
       dog  short -0.006154 -0.923061
```

### 2.8. Reshaping and pivot tables

551
Missing data

These functions are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling `sort_index()`, of course). Here is a more complex example:

```python
In [33]: columns = pd.MultiIndex.from_tuples(
   ....:     [("A", "cat"),
   ....:      ("B", "dog"),
   ....:      ("E", "cat"),
   ....:      ("A", "dog"),
   ....:     ],
   ....:     names=["exp", "animal"],
   ....: )

In [34]: index = pd.MultiIndex.from_product(
   ....:     [(["bar", "baz", "foo", "qux"), ("one", "two")], names=["first", "second"]
   ....: )

In [35]: df = pd.DataFrame(np.random.randn(8, 4), index=index, columns=columns)

In [36]: df2 = df.iloc[[0, 1, 2, 4, 5, 7]]

In [37]: df2
```

```
Out[37]:
exp animal first second
A    cat   bar one 0.895717 0.805244 -1.206412 2.565646
     dog   two 1.431256 1.340309 -1.170299 0.226169
B    cat   baz one 0.410835 0.813850 0.132003 -0.827317
     dog   two -1.413681 1.607920 1.024180 0.569605
E    cat   foo one -1.413681 2.211372 0.974466 -2.006747
     dog   two 0.875906 0.769804 -1.281247 -0.727707
A    cat   qux two -1.226825 0.769804 -1.281247 -0.727707
     dog
```

As mentioned above, `stack()` can be called with a `level` argument to select which level in the columns to stack:

```python
In [38]: df2.stack("exp")
```

```
Out[38]:
animal first second exp
A    cat   bar one 0.895717 2.565646
     dog   B -1.206412 0.805244
     dog   two 1.431256 -0.226169
     dog   B -1.170299 1.340309
B    cat   baz one 0.410835 -0.827317
     dog   B 0.132003 0.813850
     dog   two -1.413681 0.569605
     dog   B 1.024180 1.607920
E    cat   foo one -1.413681 0.569605
     dog   B 1.024180 1.607920
     dog   two 0.875906 -2.006747
     dog   B 0.974466 -2.211372
```

(continues on next page)
Unstacking can result in missing values if subgroups do not have the same set of labels. By default, missing values will be replaced with the default fill value for that data type, NaN for float, NaT for datetimelike, etc. For integer types, by default data will converted to float and missing values will be set to NaN.

Alternatively, unstack takes an optional fill_value argument, for specifying the value of missing data.
With a MultiIndex

Unstacking when the columns are a `MultiIndex` is also careful about doing the right thing:

```
In [44]: df[:3].unstack(0)
Out[44]:
 exp     A       B  A
animal  cat  dog  cat  dog
first   bar  baz  bar  baz  bar  baz
second  one 0.8957  0.4108  0.8052  0.8139 -1.2064  0.1320  2.5656
        two 1.4313   NaN    1.3403 -1.2064  0.1320  2.5656 -0.8273

In [45]: df2.unstack(1)
Out[45]:
 exp     A       B  A
animal  cat  dog  cat  dog
second  one  two  one  two  one  two  one  two  one  two
first   bar  0.8957  1.4313  0.8052  1.3403 -1.2064 -1.1703  2.5656 -0.2262
       baz  0.4108   NaN    0.8139  0.1320  2.5656 -0.8273  NaN    NaN
       foo -1.4137   0.8759 -2.2113  1.0241  0.9745  0.5696 -2.0067  NaN
       qux   NaN   -1.2268  NaN    0.7698  NaN    NaN    NaN    NaN
```
2.8.3 Reshaping by melt

The top-level `melt()` function and the corresponding `DataFrame.melt()` are useful to massage a `DataFrame` into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are “unpivoted” to the row axis, leaving just two non-identifier columns, “variable” and “value”. The names of those columns can be customized by supplying the `var_name` and `value_name` parameters.

For instance,

```python
In [46]: cheese = pd.DataFrame(
    ....:     {
    ....:         "first": ["John", "Mary"],
    ....:         "last": ["Doe", "Bo"],
    ....:         "height": [5.5, 6.0],
    ....:         "weight": [130, 150],
    ....:     }
    ....: )
    ....:

In [47]: cheese
Out[47]:
   first  last  height  weight
0  John   Doe     5.5     130
1  Mary    Bo     6.0     150

In [48]: cheese.melt(id_vars=['first', 'last'])
Out[48]:
   first  last  variable  value
0  John   Doe    height     5.5
1  Mary    Bo    height     6.0
2  John   Doe    weight    130.0
3  Mary    Bo    weight    150.0
```

(continues on next page)
When transforming a DataFrame using `melt()`, the index will be ignored. The original index values can be kept around by setting the `ignore_index` parameter to `False` (default is `True`). This will however duplicate them.

New in version 1.1.0.

Another way to transform is to use the `wide_to_long()` panel data convenience function. It is less flexible than `melt()`, but more user-friendly.
In [55]:
def = pd.DataFrame(
    
    "A1970": {0: "a", 1: "b", 2: "c"},
    "A1980": {0: "d", 1: "e", 2: "f"},
    "B1970": {0: 2.5, 1: 1.2, 2: 0.7},
    "B1980": {0: 3.2, 1: 1.3, 2: 0.1},
    "X": dict(zip(range(3), np.random.randn(3)))).

In [56]:
def["id"] = def.index

In [57]:
def

Out[57]:

0 a d 2.5 3.2 -0.121306 0
1 b e 1.2 1.3 -0.097883 1
2 c f 0.7 0.1 0.695775 2

In [58]:
pd.wide_to_long(def, ["A", "B"], i="id", j="year")

Out[58]:

X A B
id year
0 1970 -0.121306 a 2.5
1 1970 -0.097883 b 1.2
2 1970 0.695775 c 0.7
0 1980 -0.121306 d 3.2
1 1980 -0.097883 e 1.3
2 1980 0.695775 f 0.1

2.8.4 Combining with stats and GroupBy

It should be no shock that combining `pivot()` / `stack()` / `unstack()` with GroupBy and the basic Series and DataFrame statistical functions can produce some very expressive and fast data manipulations.

In [59]:
df

Out[59]:

exp animal A B A
first second cat dog cat dog
bar one 0.895717 0.805244 -1.206412 2.565646
two 1.431256 1.340309 -1.170299 -0.226169
baz one 0.410835 0.813850 0.132003 -0.827317
two -0.076467 -1.187678 1.130127 -1.436737
foo one -1.413681 1.607920 1.024180 0.569605
two 0.875906 -2.211372 0.974466 -2.006747
qux one -0.410001 -0.078638 0.545952 -1.219217
two -1.226825 0.769804 -1.281247 -0.727707

In [60]:
df.stack().mean(1).unstack()
# same result, another way

```python
In [61]: df.groupby(level=1, axis=1).mean()
```

```
Out[61]:
animal    cat   dog
first   second
bar  one  -0.155347  1.685445
      two   0.130479   0.557070
baz  one   0.271419 -0.006733
      two   0.526830 -1.312207
foo  one  -0.194750  1.088763
      two   0.925186 -2.109060
qux  one   0.067976 -0.648927
      two  -1.254036   0.021048
```

```python
In [62]: df.stack().groupby(level=1).mean()
```

```
Out[62]:
exp    A   B
second
one   0.071448  0.455513
two  -0.424186 -0.204486
```

```python
In [63]: df.mean().unstack(0)
```

```
Out[63]:
exp    A   B
animal
cat   0.060843  0.018596
dog  -0.413580  0.232430
```

## 2.8.5 Pivot tables

While `pivot()` provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides `pivot_table()` for pivoting with aggregation of numeric data.

The function `pivot_table()` can be used to create spreadsheet-style pivot tables. See the `cookbook` for some advanced strategies.

It takes a number of arguments:

- `data`: a DataFrame object.
- `values`: a column or a list of columns to aggregate.
• **index**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

• **columns**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

• **aggfunc**: function to use for aggregation, defaulting to `numpy.mean`.

Consider a data set like this:

```python
In [64]: import datetime
In [65]: df = pd.DataFrame(
   ...:     {
   ...:         "A": ["one", "one", "two", "three"] * 6,
   ...:         "B": ["A", "B", "C"] * 8,
   ...:         "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 4,
   ...:         "D": np.random.randn(24),
   ...:         "E": np.random.randn(24),
   ...:         "F": [datetime.datetime(2013, i, 1) for i in range(1, 13)]
   ...:          + [datetime.datetime(2013, i, 15) for i in range(1, 13)],
   ...:     })
In [66]: df
Out[66]:
   A  B  C  D       E  F
0  one A  foo  0.341734 -0.317441 2013-01-01
1  one B  foo  0.959726 -1.236269 2013-02-01
2  two C  foo -1.110336  0.896171 2013-03-01
3  three A bar -0.619976 -0.487602 2013-04-01
4   one B bar  0.149748 -0.082240 2013-05-01
..   ... .. ... ... ... ...
19 three B  foo  0.690579 -2.213588 2013-08-15
20 one C  foo  0.995761  1.063327 2013-09-15
21 one A  bar  2.396780  1.266143 2013-10-15
22 two B  bar  0.014871  0.299368 2013-11-15
23 three C  bar  3.357427 -0.863838 2013-12-15
[24 rows x 6 columns]
```

We can produce pivot tables from this data very easily:

```python
In [67]: pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
Out[67]:
   C    bar    foo
A  B
one A  1.120915 -0.514058
   B  -0.338421  0.002759
   C  -0.538846  0.699535
three A -1.181568    NaN
   B   NaN     0.433512
   C  0.588783   NaN
two A   NaN     1.000985
```

(continues on next page)
The result object is a DataFrame having potentially hierarchical indexes on the rows and columns. If the values column name is not given, the pivot table will include all of the data that can be aggregated in an additional level of hierarchy in the columns:

```python
In [70]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])
Out[70]:
   D         E
A one three two one
  bar foo bar foo bar foo
B A 1.120915 -0.514058 1.393057 -0.021605
     B -0.338421 0.002759 0.684149 -0.551692
     C -0.538846 0.699535 -0.988442 0.747859
three A -1.181568 NaN 0.961289 NaN
    B NaN 0.433512 NaN -1.064372
    C 0.588783 NaN -0.131830 NaN
two A NaN 1.000985 NaN 0.064245
   B 0.158248 NaN -0.097147 NaN
```

(continues on next page)
Also, you can use `Grouper` for index and columns keywords. For detail of `Grouper`, see Grouping with a Grouper specification.

```
In [71]: pd.pivot_table(df, values="D", index=pd.Grouper(freq="M", key="F"), columns="C")
Out[71]:
C         bar         foo
F
2013-01-31 NaN -0.514058
2013-02-28 NaN  0.002759
2013-03-31 NaN  0.176180
2013-04-30 -1.181568 NaN
2013-05-31 -0.338421  NaN
2013-06-30 -0.538846  NaN
2013-07-31 NaN  1.000985
2013-08-31 NaN  0.433512
2013-09-30 NaN  0.699535
2013-10-31  1.120915 NaN
2013-11-30 NaN  0.158248
2013-12-31 NaN  0.588783
```

You can render a nice output of the table omitting the missing values by calling `to_string()` if you wish:

```
In [72]: table = pd.pivot_table(df, index=["A", "B"], columns=["C"])
In [73]: print(table.to_string(na_rep=""))
```

```
D     E
bar   foo   bar   foo
A
  one   A  1.120915 -0.514058  1.393057 -0.021605
     B -0.338421  0.002759  0.684149 -0.551692
     C -0.538846  0.699535 -0.988442  0.747859
  three  A -1.181568  0.961289
         B  0.433512 -1.064372
         C  0.588783 -0.131830
  two   A  1.000985  0.064245
     B  0.158248 -0.097147
     C  0.176180  0.436241
```

Note that `pivot_table()` is also available as an instance method on DataFrame, i.e. `DataFrame.pivot_table()`.
Adding margins

If you pass margins=True to pivot_table(), special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
In [74]: table = df.pivot_table(index=["A", "B"], columns="C", margins=True, aggfunc=np.std)
```

```
In [75]: table
Out[75]:
```

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th></th>
<th></th>
<th>E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bar</td>
<td>foo</td>
<td>All</td>
<td>bar</td>
<td>foo</td>
<td>All</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1.804346</td>
<td>1.210272</td>
<td>1.569879</td>
<td>0.179483</td>
<td>0.418374</td>
<td>0.858005</td>
</tr>
<tr>
<td>C</td>
<td>0.690376</td>
<td>1.353355</td>
<td>0.898998</td>
<td>1.083825</td>
<td>0.968138</td>
<td>1.101401</td>
</tr>
<tr>
<td>C</td>
<td>0.273641</td>
<td>0.418926</td>
<td>0.771139</td>
<td>1.689271</td>
<td>0.446140</td>
<td>1.422136</td>
</tr>
<tr>
<td>three</td>
<td>0.794212</td>
<td>0.794212</td>
<td>2.049040</td>
<td>NaN</td>
<td>2.049040</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3.915454</td>
<td>NaN</td>
<td>3.915454</td>
<td>1.035215</td>
<td>NaN</td>
<td>1.035215</td>
</tr>
<tr>
<td>C</td>
<td>NaN</td>
<td>0.363548</td>
<td>0.363548</td>
<td>NaN</td>
<td>1.625237</td>
<td>1.625237</td>
</tr>
<tr>
<td>C</td>
<td>3.915454</td>
<td>NaN</td>
<td>3.915454</td>
<td>1.035215</td>
<td>NaN</td>
<td>1.035215</td>
</tr>
<tr>
<td>two</td>
<td>A</td>
<td>NaN</td>
<td>0.442998</td>
<td>0.442998</td>
<td>NaN</td>
<td>0.447104</td>
</tr>
<tr>
<td>C</td>
<td>0.202765</td>
<td>NaN</td>
<td>0.202765</td>
<td>0.560757</td>
<td>NaN</td>
<td>0.560757</td>
</tr>
<tr>
<td>C</td>
<td>NaN</td>
<td>1.819408</td>
<td>1.819408</td>
<td>NaN</td>
<td>0.650439</td>
<td>0.650439</td>
</tr>
<tr>
<td>C</td>
<td>1.556686</td>
<td>0.952552</td>
<td>1.246608</td>
<td>1.250924</td>
<td>0.899904</td>
<td>1.059389</td>
</tr>
</tbody>
</table>

Additionally, you can call DataFrame.stack() to display a pivoted DataFrame as having a multi-level index:

```
In [76]: table.stack()
Out[76]:
```

```
<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th></th>
<th>E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1.569879</td>
<td>0.858005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.804346</td>
<td>0.179483</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.210272</td>
<td>0.418374</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.690376</td>
<td>1.083825</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.898998</td>
<td>1.101401</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.273641</td>
<td>1.689271</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.418926</td>
<td>0.446140</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>three</td>
<td>0.794212</td>
<td>0.794212</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3.915454</td>
<td>1.035215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>NaN</td>
<td>0.363548</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3.915454</td>
<td>1.035215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>NaN</td>
<td>0.442998</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.202765</td>
<td>0.560757</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>NaN</td>
<td>1.819408</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.556686</td>
<td>1.246608</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.952552</td>
<td>1.250924</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>two</td>
<td>1.819408</td>
<td>0.650439</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.246608</td>
<td>1.059389</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1.556686</td>
<td>1.250924</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.952552</td>
<td>0.899904</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[24 rows x 2 columns]
2.8.6 Cross tabulations

Use crosstab() to compute a cross-tabulation of two (or more) factors. By default crosstab() computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- **index**: array-like, values to group by in the rows.
- **columns**: array-like, values to group by in the columns.
- **values**: array-like, optional, array of values to aggregate according to the factors.
- **aggfunc**: function, optional, If no values array is passed, computes a frequency table.
- **rownames**: sequence, default None, must match number of row arrays passed.
- **colnames**: sequence, default None, if passed, must match number of column arrays passed.
- **margins**: boolean, default False, Add row/column margins (subtotals)
- **normalize**: boolean, {'all', 'index', 'columns'}, or {0,1}, default False. Normalize by dividing all values by the sum of values.

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

For example:

```python
In [77]: foo, bar, dull, shiny, one, two = "foo", "bar", "dull", "shiny", "one", "two"

In [78]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [79]: b = np.array([one, one, two, one, two, one], dtype=object)

In [80]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [81]: pd.crosstab(a, [b, c], rownames=["a"], colnames=["b", "c"])
```

```python
Out[81]:
   b  one  two
c  dull  shiny
dull  1    0   0
  shiny  1    1   0
a
  bar  1 0 0 1
  foo  2 1 1 0
```

If crosstab() receives only two Series, it will provide a frequency table.

```python
In [82]: df = pd.DataFrame(  
...:   {"A": [1, 2, 2, 2, 2], "B": [3, 3, 4, 4, 4], "C": [1, 1, np.nan, 1, 1]}  
...: )

In [83]: df
```

```python
Out[83]:
   A  B  C
0  1  3  1.0
1  2  3  1.0
2  2  4  NaN
3  2  4  1.0
```

(continues on next page)
In [84]: pd.crosstab(df["A"], df["B"])
Out[84]:
    3  4
A
1  1  0
2  1  3

crosstab() can also be implemented to Categorical data.

In [85]: foo = pd.Categorical(["a", "b"], categories=["a", "b", "c"])
In [86]: bar = pd.Categorical(["d", "e"], categories=["d", "e", "f"])  

In [87]: pd.crosstab(foo, bar)
Out[87]:
       col_0  d  e  f
row_0   a  1  0  0
        b  0  1  0
        c  0  0  0

If you want to include all of data categories even if the actual data does not contain any instances of a particular category, you should set dropna=False.

For example:

In [88]: pd.crosstab(foo, bar, dropna=False)
Out[88]:
       col_0  d  e  f
row_0   a  1  0  0
        b  0  1  0
        c  0  0  0

Normalization

Frequency tables can also be normalized to show percentages rather than counts using the normalize argument:

In [89]: pd.crosstab(df["A"], df["B"], normalize=True)
Out[89]:
   B  3  4
A
1  0.2  0.0
2  0.2  0.6

normalize can also normalize values within each row or within each column:

In [90]: pd.crosstab(df["A"], df["B"], normalize="columns")
Out[90]:
   B  3  4
A
1  0.2  0.0
2  0.2  0.6
crosstab() can also be passed a third Series and an aggregation function (aggfunc) that will be applied to the values of the third Series within each group defined by the first two Series:

```
In [91]: pd.crosstab(df['A'], df['B'], values=df['C'], aggfunc=np.sum)
Out[91]:
   B    3    4
  A
  1   1.0  NaN
  2   1.0  2.0
```

Adding margins

Finally, one can also add margins or normalize this output.

```
In [92]: pd.crosstab(
   ....:     df['A'], df['B'], values=df['C'], aggfunc=np.sum, normalize=True,
   ....:     margins=True)
   ....: )
   ....:
Out[92]:
   B    3    4  All
  A
  1  0.25  0.0  0.25
  2  0.25  0.5  0.75
 All 0.50  0.5  1.00
```

### 2.8.7 Tiling

The cut() function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

```
In [93]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])
In [94]: pd.cut(ages, bins=3)
Out[94]:
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (26.667, 43.333], (43.333, 60.0], (43.333, 60.0]]
Categories (3, interval[float64, right]): [(9.95, 26.667] < (26.667, 43.333] < (43.333, 60.0]]
```

If the bins keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [95]: c = pd.cut(ages, bins=3)
In [96]: c
Out[96]:
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]]
Categories (3, interval[int64, right]): [(0, 18] < (18, 35] < (35, 70]]
```
If the bins keyword is an `IntervalIndex`, then these will be used to bin the passed data:

```python
pd.cut([25, 20, 50], bins=c.categories)
```

### 2.8.8 Computing indicator / dummy variables

To convert a categorical variable into a “dummy” or “indicator” `DataFrame`, for example a column in a `DataFrame` (a `Series`) which has k distinct values, can derive a `DataFrame` containing k columns of 1s and 0s using `get_dummies()`:

```python
In [97]: df = pd.DataFrame({"key": list("bbacab"), "data1": range(6)})
In [98]: pd.get_dummies(df["key"])
Out[98]:
         a  b  c  
0     0  0  1  
1     0  1  0  
2     1  0  0  
3     0  0  1  
4     1  0  0  
5     0  1  0  
```

Sometimes it’s useful to prefix the column names, for example when merging the result with the original `DataFrame`:

```python
In [99]: dummies = pd.get_dummies(df["key"], prefix="key")
In [100]: dummies
Out[100]:
         key_a  key_b  key_c  
0     0.0  0.0  1.0  
1     0.0  1.0  0.0  
2     1.0  0.0  0.0  
3     0.0  0.0  1.0  
4     1.0  0.0  0.0  
5     0.0  1.0  0.0  
```

This function is often used along with discretization functions like `cut()`:

```python
In [102]: values = np.random.randn(10)
In [103]: values
Out[103]:
array([ 0.4082, -1.0481, -0.0257, -0.9884,  0.0941,  1.2627,  1.29 ,
   -0.8771, -0.0497,  0.8576]),
```
In [104]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [105]: pd.get_dummies(pd.cut(values, bins))

Out[105]:

<table>
<thead>
<tr>
<th>bins</th>
<th>(0.0, 0.2]</th>
<th>(0.2, 0.4]</th>
<th>(0.4, 0.6]</th>
<th>(0.6, 0.8]</th>
<th>(0.8, 1.0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

See also `Series.str.get_dummies`.

`get_dummies()` also accepts a `DataFrame`. By default all categorical variables (categorical in the statistical sense, those with object or categorical dtype) are encoded as dummy variables.

In [106]: df = pd.DataFrame({"A": ["a", "b", "a"], "B": ["c", "c", "b"], "C": [1, 2, 3]})

In [107]: pd.get_dummies(df)

Out[107]:

<table>
<thead>
<tr>
<th>C</th>
<th>A_a</th>
<th>A_b</th>
<th>B_b</th>
<th>B_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

All non-object columns are included untouched in the output. You can control the columns that are encoded with the `columns` keyword.

In [108]: pd.get_dummies(df, columns=["A"])

Out[108]:

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>A_a</th>
<th>A_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Notice that the B column is still included in the output, it just hasn’t been encoded. You can drop B before calling `get_dummies` if you don’t want to include it in the output.

As with the `Series` version, you can pass values for the `prefix` and `prefix_sep`. By default the column name is used as the prefix, and _ as the prefix separator. You can specify `prefix` and `prefix_sep` in 3 ways:

- string: Use the same value for `prefix` or `prefix_sep` for each column to be encoded.
- list: Must be the same length as the number of columns being encoded.
- dict: Mapping column name to prefix.
In [109]: simple = pd.get_dummies(df, prefix="new_prefix")

In [110]: simple
Out[110]:
<table>
<thead>
<tr>
<th></th>
<th>new_prefix_a</th>
<th>new_prefix_b</th>
<th>new_prefix_b</th>
<th>new_prefix_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In [111]: from_list = pd.get_dummies(df, prefix=['from_A', "from_B"])

In [112]: from_list
Out[112]:
<table>
<thead>
<tr>
<th></th>
<th>from_A_a</th>
<th>from_A_b</th>
<th>from_B_b</th>
<th>from_B_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In [113]: from_dict = pd.get_dummies(df, prefix={'A': "from_A", "B": "from_B"})

In [114]: from_dict
Out[114]:
<table>
<thead>
<tr>
<th></th>
<th>from_A_a</th>
<th>from_A_b</th>
<th>from_B_b</th>
<th>from_B_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sometimes it will be useful to only keep k-1 levels of a categorical variable to avoid collinearity when feeding the result to statistical models. You can switch to this mode by turn on drop_first.

In [115]: s = pd.Series(list("abcaa"))

In [116]: pd.get_dummies(s)
Out[116]:
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In [117]: pd.get_dummies(s, drop_first=True)
Out[117]:
<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When a column contains only one level, it will be omitted in the result.
In [119]: pd.get_dummies(df)
Out[119]:
   A_a  B_a  B_b  B_c
0  1   1   0   0
1  1   0   1   0
2  1   1   0   0
3  1   0   1   0
4  1   0   0   1

In [120]: pd.get_dummies(df, drop_first=True)
Out[120]:
   B_b  B_c
0   0   0
1   1   0
2   0   0
3   1   0
4   0   1

By default new columns will have np.uint8 dtype. To choose another dtype, use the dtype argument:

In [121]: df = pd.DataFrame({"A": list("abc"), "B": [1.1, 2.2, 3.3]})

In [122]: pd.get_dummies(df, dtype=bool).dtypes
Out[122]:
   B  float64
  A_a  bool
  A_b  bool
  A_c  bool
dtype: object

2.8.9 Factorizing values

To encode 1-d values as an enumerated type use factorize():


In [124]: x
Out[124]:
   0  A
  1  A
  2  NaN
  3  B
  4  3.14
  5  inf
dtype: object

In [125]: labels, uniques = pd.factorize(x)

In [126]: labels
Out[126]: array([ 0,  0, -1,  1,  2,  3])

(continues on next page)
In [127]: uniques
Out[127]: Index(['A', 'B', 3.14, inf], dtype='object')

Note that `factorize()` is similar to `numpy.unique`, but differs in its handling of NaN:

Note: The following `numpy.unique` will fail under Python 3 with a `TypeError` because of an ordering bug. See also here.

In [129]: pd.factorize(ser, sort=True)
Out[129]: (array([ 2, 2, -1, 3, 0, 1]), Index([3.14, inf, 'A', 'B'], dtype='object'))
In [130]: np.unique(ser, return_inverse=True)[::-1]
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
Input In [130], in <cell line: 1>()
----> 1 np.unique(ser, return_inverse=True)[::-1]
File <__array_function__ internals>:180, in unique(*args, **kwargs)
File /opt/conda/envs/pandas/lib/python3.8/site-packages/numpy/lib/arraysetops.py:272, in _unique(ar, return_index, return_inverse, return_counts, axis)
   270     ar = np.asanyarray(ar)
   271     if axis is None:
--> 272         perm = ar.argsort(kind='mergesort' if return_index else 'quicksort')
   273         aux = ar[perm]
   275     # axis was specified and not None

File /opt/conda/envs/pandas/lib/python3.8/site-packages/numpy/lib/arraysetops.py:330, in _uniqueid(ar, return_index, return_inverse, return_counts)
   327     if optional_indices:
   329         if optional_indices:
--> 330             perm = ar.argsort(kind='mergesort' if return_index else 'quicksort')
   331             aux = ar[perm]
   332 else:

TypeError: '<' not supported between instances of 'float' and 'str'

Note: If you just want to handle one column as a categorical variable (like R’s factor), you can use `df["cat_col"] = pd.Categorical(df["col"])` or `df["cat_col"] = df["col"].astype("category")`. For full docs on `Categorical`, see the `Categorical introduction` and the `API documentation`.  

570 Chapter 2. User Guide
2.8.10 Examples

In this section, we will review frequently asked questions and examples. The column names and relevant column values are named to correspond with how this DataFrame will be pivoted in the answers below.

```
In [131]: np.random.seed([3, 1415])
In [132]: n = 20
In [133]: cols = np.array(['key', 'row', 'item', 'col'])
In [134]: df = cols + pd.DataFrame(
    ....:     (np.random.randint(5, size=(n, 4)) // [2, 1, 2, 1]).astype(str)
    ....: )
In [135]: df.columns = cols
In [136]: df = df.join(pd.DataFrame(np.random.rand(n, 2).round(2)).add_prefix('val'))
In [137]: df
```

```
Out[137]:
     key row item  col  val0  val1
0   key0 row3 item1 col3  0.81  0.04
1   key1 row2 item1 col2  0.44  0.07
2   key1 row0 item1 col0  0.77  0.01
3   key0 row4 item0 col2  0.15  0.59
4   key1 row0 item2 col1  0.81  0.64
..   ...   ...   ...   ...   ...
15  key0 row3 item1 col2  0.31  0.23
16  key0 row0 item2 col3  0.13  0.45
17  key2 row2 item2 col3  0.37  0.70
18  key0 row2 item0 col4  0.37  0.70
[20 rows x 6 columns]
```

### Pivoting with single aggregations

Suppose we wanted to pivot `df` such that the `col` values are columns, `row` values are the index, and the mean of `val0` are the values? In particular, the resulting DataFrame should look like:

```
col  col0  col1  col2  col3  col4
row
row0  0.77  0.60  NaN   0.86  0.65
row2  0.13  NaN   0.39  0.50  0.25
row3  NaN  0.31  NaN   0.54  NaN
row4  NaN  0.10  0.39  0.76  0.24
```

This solution uses `pivot_table()`. Also note that `aggfunc='mean'` is the default. It is included here to be explicit.

```
In [138]: df.pivot_table(values='val0', index='row', columns='col', aggfunc='mean')
```

(continues on next page)
Note that we can also replace the missing values by using the \texttt{fill\_value} parameter.

\begin{verbatim}
In [139]: df.pivot_table(
        values="val0",
        index="row",
        columns="col",
        aggfunc="mean",
        fill_value=0,
    )
\end{verbatim}

\begin{verbatim}
Out[139]:
col col0 col1 col2 col3 col4
row
row0 0.77 0.605 NaN 0.860 0.65
row2 0.13 NaN 0.395 0.500 0.25
row3 NaN 0.310 NaN 0.545 NaN
row4 NaN 0.100 0.395 0.760 0.24
\end{verbatim}

Also note that we can pass in other aggregation functions as well. For example, we can also pass in \texttt{sum}.

\begin{verbatim}
In [140]: df.pivot_table(
        values="val0",
        index="row",
        columns="col",
        aggfunc="sum",
        fill_value=0,
    )
\end{verbatim}

\begin{verbatim}
Out[140]:
col col0 col1 col2 col3 col4
row
row0 0.77 1.21 0.00 0.86 0.65
row2 0.13 0.00 0.79 0.50 0.50
row3 0.00 0.31 0.00 1.09 0.00
row4 0.00 0.10 0.79 1.52 0.24
\end{verbatim}

Another aggregation we can do is calculate the frequency in which the columns and rows occur together a.k.a. “cross tabulation”. To do this, we can pass \texttt{size} to the \texttt{aggfunc} parameter.

\begin{verbatim}
In [141]: df.pivot_table(index="row", columns="col", fill_value=0, aggfunc="size")
\end{verbatim}

\begin{verbatim}
Out[141]:
col col0 col1 col2 col3 col4
row
row0 1 2 0 1 1
row2 1 0 2 1 2
\end{verbatim}
Pivoting with multiple aggregations

We can also perform multiple aggregations. For example, to perform both a sum and mean, we can pass in a list to the `aggfunc` argument.

```python
In [142]: df.pivot_table(
   ....:     values="val0",
   ....:     index="row",
   ....:     columns="col",
   ....:     aggfunc=["mean", "sum"],
   ....: )
```

```python
Out[142]:
```

<table>
<thead>
<tr>
<th>mean</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>col</td>
<td>col0</td>
</tr>
<tr>
<td>row</td>
<td>col0</td>
</tr>
<tr>
<td>row0</td>
<td>0.77</td>
</tr>
<tr>
<td>row2</td>
<td>0.13</td>
</tr>
<tr>
<td>row3</td>
<td>NaN</td>
</tr>
<tr>
<td>row4</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

Note to aggregate over multiple value columns, we can pass in a list to the `values` parameter.

```python
In [143]: df.pivot_table(
   ....:     values=["val0", "val1"],
   ....:     index="row",
   ....:     columns="col",
   ....:     aggfunc=["mean"],
   ....: )
```

```python
Out[143]:
```

<table>
<thead>
<tr>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>val0</td>
</tr>
<tr>
<td>val1</td>
</tr>
<tr>
<td>col</td>
</tr>
<tr>
<td>row</td>
</tr>
<tr>
<td>row0</td>
</tr>
<tr>
<td>row2</td>
</tr>
<tr>
<td>row3</td>
</tr>
<tr>
<td>row4</td>
</tr>
</tbody>
</table>
```

Note to subdivide over multiple columns we can pass in a list to the `columns` parameter.

```python
In [144]: df.pivot_table(
   ....:     values=["val0"],
   ....:     index="row",
   ....:     columns=["item", "col"],
   ....:     aggfunc=["mean"],
   ....: )
```

Note to subdivide over multiple columns we can pass in a list to the `columns` parameter.
2.8.11 Exploding a list-like column

New in version 0.25.0.

Sometimes the values in a column are list-like.

```python
In [145]: keys = ["panda1", "panda2", "panda3"]

In [146]: values = [["eats", "shoots"], ["shoots", "leaves"], ["eats", "leaves"]]

In [147]: df = pd.DataFrame({"keys": keys, "values": values})

In [148]: df
```

We can ‘explode’ the values column, transforming each list-like to a separate row, by using `explode()`. This will replicate the index values from the original row:

```python
In [149]: df["values"].explode()
```

You can also explode the column in the `DataFrame`.

```python
In [150]: df.explode("values")
```
Series.explode() will replace empty lists with np.nan and preserve scalar entries. The dtype of the resulting Series is always object.

In [151]: s = pd.Series([[1, 2, 3], "foo", [], ["a", "b"]])

In [152]: s
Out[152]:
0    [1, 2, 3]
1      foo
2          []
3     [a, b]
dtype: object

In [153]: s.explode()
Out[153]:
0    0
0    1
0    2
0    3
1    foo
2   NaN
3     a
3     b
dtype: object

Here is a typical usecase. You have comma separated strings in a column and want to expand this.

In [154]: df = pd.DataFrame([{"var1": "a,b,c", "var2": 1}, {"var1": "d,e,f", "var2": 2}])

In [155]: df
Out[155]:
   var1     var2
0  a,b,c       1
1  d,e,f       2

Creating a long form DataFrame is now straightforward using explode and chained operations

In [156]: df.assign(var1=df.var1.str.split(",")).explode("var1")
Out[156]:
   var1     var2
0    a       1
0    b       1
0    c       1
1    d       2
1    e       2
1    f       2

2.8. Reshaping and pivot tables 575
2.9 Working with text data

2.9.1 Text data types

New in version 1.0.0.

There are two ways to store text data in pandas:

1. object dtype NumPy array.
2. StringDtype extension type.

We recommend using StringDtype to store text data.

Prior to pandas 1.0, object dtype was the only option. This was unfortunate for many reasons:

1. You can accidentally store a mixture of strings and non-strings in an object dtype array. It's better to have a dedicated dtype.
2. object dtype breaks dtype-specific operations like DataFrame.select_dtypes(). There isn't a clear way to select just text while excluding non-text but still object-dtype columns.
3. When reading code, the contents of an object dtype array is less clear than 'string'.

Currently, the performance of object dtype arrays of strings and arrays.StringArray are about the same. We expect future enhancements to significantly increase the performance and lower the memory overhead of StringArray.

**Warning:** StringArray is currently considered experimental. The implementation and parts of the API may change without warning.

For backwards-compatibility, object dtype remains the default type we infer a list of strings to

```
In [1]: pd.Series(["a", "b", "c"])
Out[1]:
0   a
1   b
2   c
dtype: object
```

To explicitly request string dtype, specify the dtype

```
In [2]: pd.Series(["a", "b", "c"], dtype="string")
Out[2]:
0   a
1   b
2   c
dtype: string
```

```
In [3]: pd.Series(["a", "b", "c"], dtype=pd.StringDtype())
Out[3]:
0   a
1   b
2   c
dtype: string
```

Or astype after the Series or DataFrame is created
In [4]: s = pd.Series(["a", "b", "c"])

In [5]: s
Out[5]:
0   a
1   b
2   c
dtype: object

In [6]: s.astype("string")
Out[6]:
0   a
1   b
2   c
dtype: string

Changed in version 1.1.0.
You can also use `StringDtype"string"` as the dtype on non-string data and it will be converted to string dtype:

In [7]: s = pd.Series(["a", 2, np.nan], dtype="string")

In [8]: s
Out[8]:
0   a
1   2
2   <NA>
dtype: string

In [9]: type(s[1])
Out[9]: str

or convert from existing pandas data:

In [10]: s1 = pd.Series([1, 2, np.nan], dtype="Int64")

In [11]: s1
Out[11]:
0   1
1   2
2   <NA>
dtype: Int64

In [12]: s2 = s1.astype("string")

In [13]: s2
Out[13]:
0   1
1   2
2   <NA>
dtype: string

In [14]: type(s2[0])
Out[14]: str
Behavior differences

These are places where the behavior of `StringDtype` objects differ from object dtype:

1. For `StringDtype`, string accessor methods that return numeric output will always return a nullable integer dtype, rather than either int or float dtype, depending on the presence of NA values. Methods returning boolean output will return a nullable boolean dtype.

```
In [15]: s = pd.Series(["a", None, "b"], dtype="string")

In [16]: s
Out[16]:
  0    a
  1  <NA>
  2    b
dtype: string

In [17]: s.str.count("a")
Out[17]:
  0    1
  1  <NA>
  2    0
dtype: Int64

In [18]: s.dropna().str.count("a")
Out[18]:
  0    1
  2    0
dtype: Int64
```

Both outputs are Int64 dtype. Compare that with object-dtype

```
In [19]: s2 = pd.Series(["a", None, "b"], dtype="object")

In [20]: s2.str.count("a")
Out[20]:
  0    1.0
  1  NaN
  2    0.0
dtype: float64

In [21]: s2.dropna().str.count("a")
Out[21]:
  0    1
  2    0
dtype: int64
```

When NA values are present, the output dtype is float64. Similarly for methods returning boolean values.

```
In [22]: s.str.isdigit()
Out[22]:
  0  False
  1  <NA>
  2  False
```

(continues on next page)
2. Some string methods, like `Series.str.decode()` are not available on `StringArray` because `StringArray` only holds strings, not bytes.

3. In comparison operations, `arrays.StringArray` and `Series` backed by a `StringArray` will return an object with `BooleanDtype`, rather than a `bool` dtype object. Missing values in a `StringArray` will propagate in comparison operations, rather than always comparing unequal like `numpy.nan`.

Everything else that follows in the rest of this document applies equally to `string` and `object` dtype.

### 2.9.2 String methods

Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the `str` attribute and generally have names matching the equivalent (scalar) built-in string methods:

```python
In [24]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'], dtype='string')

In [25]: s.str.lower()
Out[25]:
   0  a
   1  b
   2  c
   3  aaba
   4  baca
   5  <NA>
   6  cat
   7  dog
   8  cat

In [26]: s.str.upper()
Out[26]:
   0  A
   1  B
   2  C
   3  AABA
   4  BACA
   5  <NA>
   6  CABA
```

(continues on next page)
The string methods on Index are especially useful for cleaning up or transforming DataFrame columns. For instance, you may have columns with leading or trailing whitespace:

```
In [32]: df = pd.DataFrame(
      .....:   np.random.randn(3, 2), columns=[' Column A ', ' Column B '], index=range(3)
      .....: )
      ...

In [33]: df
Out[33]:
      Column A    Column B
     0  0.469112  -0.282863
     1 -1.509059  -1.135632
     2  1.212112  -0.173215
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```
In [34]: df.columns.str.strip()
Out[34]: Index(['Column A', 'Column B'], dtype='object')

In [35]: df.columns.str.lower()
Out[35]: Index(['column a', 'column b'], dtype='object')
```

These string methods can then be used to clean up the columns as needed. Here we are removing leading and trailing whitespaces, lower casing all names, and replacing any remaining whitespaces with underscores:
In [36]: df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_")

In [37]: df

Out[37]:
    column_a  column_b
0   0.469112  -0.282863
1  -1.509059  -1.135632
2   1.212112  -0.173215

Note: If you have a Series where lots of elements are repeated (i.e. the number of unique elements in the Series is a lot smaller than the length of the Series), it can be faster to convert the original Series to one of type category and then use .str.<method> or .dt.<property> on that. The performance difference comes from the fact that, for Series of type category, the string operations are done on the .categories and not on each element of the Series.

Please note that a Series of type category with string .categories has some limitations in comparison to Series of type string (e.g. you can’t add strings to each other: s + " " + s won’t work if s is a Series of type category). Also, .str methods which operate on elements of type list are not available on such a Series.

Warning: Before v.0.25.0, the .str-accessor did only the most rudimentary type checks. Starting with v.0.25.0, the type of the Series is inferred and the allowed types (i.e. strings) are enforced more rigorously.

Generally speaking, the .str accessor is intended to work only on strings. With very few exceptions, other uses are not supported, and may be disabled at a later point.

2.9.3 Splitting and replacing strings

Methods like split return a Series of lists:

In [38]: s2 = pd.Series(\"a_b_c\", \"c_d_e\", np.nan, \"f_g_h\"), dtype=\"string\")

In [39]: s2.str.split("_")

Out[39]:
    0    [a, b, c]
    1    [c, d, e]
    2  <NA>  
    3    [f, g, h]
dtype: object

Elements in the split lists can be accessed using get or [] notation:

In [40]: s2.str.split("_").str.get(1)
Out[40]:
    0    b
    1    d
    2  <NA>
    3    g
dtype: object

In [41]: s2.str.split("_").str[1]
Out[41]:
(continues on next page)
It is easy to expand this to return a DataFrame using `expand`.

```
In [42]: s2.str.split("_", expand=True)
Out[42]:
     0  1  2
0   a  b  c
1   c  d  e
2 <NA> <NA> <NA>
3   f  g  h
```

When original `Series` has `StringDtype`, the output columns will all be `StringDtype` as well.

It is also possible to limit the number of splits:

```
In [43]: s2.str.split("_", expand=True, n=1)
Out[43]:
     0  1
0  a  b_c
1  c  d_e
2 <NA> <NA>
3   f  g_h
```

`rsplit` is similar to `split` except it works in the reverse direction, i.e., from the end of the string to the beginning of the string:

```
In [44]: s2.str.rsplit("_", expand=True, n=1)
Out[44]:
     0  1
0  a_b  c
1  c_d  e
2 <NA> <NA>
3   f_g  h
```

`replace` optionally uses regular expressions:

```
In [45]: s3 = pd.Series(
      ....:     dtype="string",
      ....:     )
      ....:
In [46]: s3
Out[46]:
     0   A
     1   B
     2   C
     3  Aaba
```
Baca
<NA>
CABA
dog
cat
dtype: string

In [47]: s3.str.replace("^\.adog", "XX-XX ", case=False, regex=True)
Out[47]:
   0   A
   1   B
   2   C
   3  XX-XX ba
   4  XX-XX ca
   5
   6  <NA>
   7  XX-XX BA
   8  XX-XX
dtype: string

Warning: Some caution must be taken when dealing with regular expressions! The current behavior is to treat single character patterns as literal strings, even when regex is set to True. This behavior is deprecated and will be removed in a future version so that the regex keyword is always respected.

Changed in version 1.2.0.

If you want literal replacement of a string (equivalent to str.replace()), you can set the optional regex parameter to False, rather than escaping each character. In this case both pat and repl must be strings:

In [48]: dollars = pd.Series(["12", "-$10", "$10,000"], dtype="string")
# These lines are equivalent
In [49]: dollars.str.replace(r"-$", "-", regex=True)
Out[49]:
   0     12
   1    -10
   2 $10,000
dtype: string
In [50]: dollars.str.replace("-$", "-", regex=False)
Out[50]:
   0     12
   1    -10
   2 $10,000
dtype: string

The replace method can also take a callable as replacement. It is called on every pat using re.sub(). The callable should expect one positional argument (a regex object) and return a string.

2.9. Working with text data
# Reverse every lowercase alphabetic word

```python
In [51]: pat = r"[a-z]+"
```

```python
In [52]: def repl(m):
    .....:     return m.group(0)[::-1]
    .....:
```

```python
In [53]: pd.Series(["foo 123", "bar baz", np.nan], dtype="string").str.replace(
    .....:     pat, repl, regex=True
    .....: )
```

```text
Out[53]:
0 oof 123
1 rab zab
2 <NA>
dtype: string
```

# Using regex groups

```python
In [54]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
```

```python
In [55]: def repl(m):
    .....:     return m.group("two").swapcase()
    .....:
```

```python
In [56]: pd.Series(["Foo Bar Baz", np.nan], dtype="string").str.replace(
    .....:     pat, repl, regex=True
    .....: )
```

```text
Out[56]:
0 bAR
1 <NA>
dtype: string
```

The `replace` method also accepts a compiled regular expression object from `re.compile()` as a pattern. All flags should be included in the compiled regular expression object.

```python
In [57]: import re
```

```python
In [58]: regex_pat = re.compile(r"^a|dog", flags=re.IGNORECASE)
```

```python
In [59]: s3.str.replace(regex_pat, "XX-XX ", regex=True)
```

```text
Out[59]:
0     A
1     B
2     C
3  XX-XX ba
4  XX-XX ca
5
6     <NA>
7  XX-XX BA
8  XX-XX
9  XX-XX t
dtype: string
```
Including a flags argument when calling `replace` with a compiled regular expression object will raise a `ValueError`.

```
In [60]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

`removeprefix` and `removesuffix` have the same effect as `str.removeprefix` and `str.removesuffix` added in Python 3.9 <https://docs.python.org/3/library/stdtypes.html#str.removeprefix>

New in version 1.4.0.

```
In [61]: s = pd.Series(["str_foo", "str_bar", "no_prefix"])
In [62]: s.str.removeprefix("str_")
Out[62]:
   str_foo
   str_bar
   no_prefix
 dtype: object

In [63]: s = pd.Series(["foo_str", "bar_str", "no_suffix"])
In [64]: s.str.removesuffix("_str")
Out[64]:
   foo
   bar
   no_suffix
 dtype: object
```

## 2.9.4 Concatenation

There are several ways to concatenate a `Series` or `Index`, either with itself or others, all based on `cat()`, resp. `Index.str.cat`.

### Concatenating a single Series into a string

The content of a `Series` (or `Index`) can be concatenated:

```
In [65]: s = pd.Series(["a", "b", "c", "d"], dtype="string")
In [66]: s.str.cat(sep="")
Out[66]: 'a,b,c,d'
```

If not specified, the keyword `sep` for the separator defaults to the empty string, `sep='':`

```
In [67]: s.str.cat()
Out[67]: 'abcd'
```

By default, missing values are ignored. Using `na_rep`, they can be given a representation:

```
In [68]: t = pd.Series(["a", "b", np.nan, "d"], dtype="string")
In [69]: t.str.cat(sep="")
```

(continues on next page)
Concatenating a Series and something list-like into a Series

The first argument to `cat()` can be a list-like object, provided that it matches the length of the calling Series (or Index).

```python
In [71]: s.str.cat(['A', 'B', 'C', 'D'])
Out[71]:
0  aA
1  bB
2  cC
3  dD
```

Missing values on either side will result in missing values in the result as well, unless `na_rep` is specified:

```python
In [72]: s.str.cat(t)
Out[72]:
0  aa
1  bb
2  <NA>
3  dd
```

```python
In [73]: s.str.cat(t, na_rep='-')
Out[73]:
0  aa
1  bb
2  -
3  dd
```

Concatenating a Series and something array-like into a Series

The parameter `others` can also be two-dimensional. In this case, the number or rows must match the lengths of the calling Series (or Index).

```python
In [74]: d = pd.concat([t, s], axis=1)
```

```python
In [75]: s
Out[75]:
0  a
1  b
2  c
3  d
```

(continues on next page)
Concatenating a Series and an indexed object into a Series, with alignment

For concatenation with a Series or DataFrame, it is possible to align the indexes before concatenation by setting the join-keyword.

```python
In [78]: u = pd.Series(["b", "d", "a", "c"], index=[1, 3, 0, 2], dtype="string")

In [79]: s
Out[79]:
  0 a
  1 b
  2 c
  3 d
dtype: string

In [80]: u
Out[80]:
  1 b
  3 d
  0 a
  2 c
dtype: string

In [81]: s.str.cat(u)
Out[81]:
   0 aa
   1 bb
   2 cc
   3 dd
dtype: string

In [82]: s.str.cat(u, join="left")
Out[82]:
   0 aa
```
Warning: If the join keyword is not passed, the method `cat()` will currently fall back to the behavior before version 0.23.0 (i.e. no alignment), but a `FutureWarning` will be raised if any of the involved indexes differ, since this default will change to `join='left'` in a future version.

The usual options are available for join (one of 'left', 'outer', 'inner', 'right'). In particular, alignment also means that the different lengths do not need to coincide anymore.

```python
In [83]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4], dtype='string')
In [84]: s
Out[84]:
0 a
1 b
2 c
3 d
dtype: string
In [85]: v
Out[85]:
-1 z
0 a
1 b
3 d
4 e
dtype: string
In [86]: s.str.cat(v, join="left", na_rep="-")
Out[86]:
  0  aa
  1  bb
  2  c-
  3  dd
dtype: string
In [87]: s.str.cat(v, join="outer", na_rep="-")
Out[87]:
  -1  -z
   0  aa
   1  bb
   2  c-
   3  dd
   4  -e
dtype: string
```

The same alignment can be used when `others` is a `DataFrame`:
In [88]: f = d.loc[[3, 2, 1, 0], :]

In [89]: s
Out[89]:
0  a
1  b
2  c
3  d
dtype: string

In [90]: f
Out[90]:
0  1
3  d  d
2  <NA>  c
1  b  b
0  a  a

In [91]: s.str.cat(f, join="left", na_rep="-")
Out[91]:
0  aaa
1  bbb
2  c-c
3  ddd
dtype: string

Concatenating a Series and many objects into a Series

Several array-like items (specifically: Series, Index, and 1-dimensional variants of np.ndarray) can be combined in a list-like container (including iterators, dict-views, etc.).
All elements without an index (e.g. np.ndarray) within the passed list-like must match in length to the calling Series (or Index), but Series and Index may have arbitrary length (as long as alignment is not disabled with join=None):

In [95]: v
Out[95]:
-1   z
  0   a
  1   b
  3   d
  4   e
dtype: string

In [96]: s.str.cat([v, u, u.to_numpy()], join="outer", na_rep="-")
Out[96]:
-1  -z--
 0   aaab
 1   bbbd
 2   c-ca
 3   dddc
 4   -e--
dtype: string

If using join='right’ on a list-like of others that contains different indexes, the union of these indexes will be used as the basis for the final concatenation:

In [97]: u.loc[[3]]
Out[97]:
 3   d
dtype: string

In [98]: v.loc[[-1, 0]]
Out[98]:
-1   z
 0   a
dtype: string

In [99]: s.str.cat([u.loc[[3]], v.loc[[-1, 0]]], join="right", na_rep="-")
Out[99]:
  3   dd-
-1  --z
 0   a-a
dtype: string
2.9.5 Indexing with .str

You can use [] notation to directly index by position locations. If you index past the end of the string, the result will be a NaN.

```
In [100]: s = pd.Series(
      ......:     ['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'],
      ......:     dtype='string' )
      ......: 

In [101]: s.str[0]
Out[101]:
0    A
1    B
2    C
3    A
4    B
5   <NA>
6    C
7    d
8    c
dtype: string

In [102]: s.str[1]
Out[102]:
0   <NA>
1   <NA>
2   <NA>
3    a
4    a
5  <NA>
6    A
7    o
8    a
dtype: string
```

2.9.6 Extracting substrings

Extract first match in each subject (extract)

```
Warning: Before version 0.23, argument expand of the extract method defaulted to False. When expand=False, expand returns a Series, Index, or DataFrame, depending on the subject and regular expression pattern. When expand=True, it always returns a DataFrame, which is more consistent and less confusing from the perspective of a user. expand=True has been the default since version 0.23.0.

The extract method accepts a regular expression with at least one capture group.

Extracting a regular expression with more than one group returns a DataFrame with one column per group.
```
Elements that do not match return a row filled with NaN. Thus, a Series of messy strings can be “converted” into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating get() to access tuples or re.match objects. The dtype of the result is always object, even if no match is found and the result only contains NaN.

Named groups like

```
In [104]: pd.Series(["a1", "b2", "c3"], dtype="string").str.extract(
            r"(?P<letter>[ab])(?P<digit>\d)", expand=False)
```

```
Out[104]:
   letter  digit
0      a      1
1      b      2
2      <NA>  <NA>
```

and optional groups like

```
In [105]: pd.Series(["a1", "b2", "3"], dtype="string").str.extract(r"([ab])?\d", expand=False)
```

```
Out[105]:
0      a      1
1      b      2
2      <NA>    3
```

can also be used. Note that any capture group names in the regular expression will be used for column names; otherwise capture group numbers will be used.

Extracting a regular expression with one group returns a DataFrame with one column if expand=True.

```
In [106]: pd.Series(["a1", "b2", "c3"], dtype="string").str.extract(r"[ab](\d)", expand=True)
```

```
Out[106]:
0      1
1      2
2      <NA>
```

It returns a Series if expand=False.
Calling on an Index with a regex with exactly one capture group returns a DataFrame with one column if expand=True.

```
In [108]: s = pd.Series(['a1', 'b2', 'c3', ['A11', 'B22', 'C33'], dtype='string'])
```

```
In [109]: s
Out[109]:
A11   a1
B22   b2
C33   c3
dtype: string
```

It returns an Index if expand=False.

```
In [110]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[110]:
   letter
0   A
1   B
2   C
```

It raises ValueError if expand=False.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

The table below summarizes the behavior of `extract(expand=False)` (input subject in first column, number of groups in regex in first row)

<table>
<thead>
<tr>
<th></th>
<th>1 group</th>
<th>&gt;1 group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Index</td>
<td>ValueError</td>
</tr>
<tr>
<td>Series</td>
<td>Series</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>
Extract all matches in each subject (extractall)

Unlike extract (which returns only the first match),

```
In [113]: s = pd.Series(["a1a2", "b1", "c1"], index="A", "B", "C", dtype="string")

In [114]: s
Out[114]:
   A    a1a2
   B     b1
   C     c1
dtype: string

In [115]: two_groups = "(?P<letter>[a-z])(?P<digit>[0-9])"

In [116]: s.str.extract(two_groups, expand=True)
Out[116]:
   letter digit
   A    a    1
   B    b    1
   C    c    1
```

the extractall method returns every match. The result of extractall is always a DataFrame with a MultiIndex on its rows. The last level of the MultiIndex is named match and indicates the order in the subject.

```
In [117]: s.str.extractall(two_groups)
Out[117]:
   letter digit
   match
   A   0   a    1
      1   a    2
   B   0   b    1
   C   0   c    1
```

When each subject string in the Series has exactly one match,

```
In [118]: s = pd.Series(["a3", "b3", "c2"], dtype="string")

In [119]: s
Out[119]:
   0    a3
   1    b3
   2    c2
dtype: string
```

then extractall(pat).xs(0, level='match') gives the same result as extract(pat).

```
In [120]: extract_result = s.str.extract(two_groups, expand=True)

In [121]: extract_result
Out[121]:
   letter digit
   match
   0    a    3
   1    b    3
```

(continues on next page)
In [122]: extractall_result = s.str.extractall(two_groups)

In [123]: extractall_result
Out[123]:
   letter digit
0  0  a  3
1  0  b  3
2  0  c  2

In [124]: extractall_result.xs(0, level="match")
Out[124]:
   letter digit
0  0  a  3
1  0  b  3
2  0  c  2

Index also supports .str.extractall. It returns a DataFrame which has the same result as a Series.str.extractall with a default index (starts from 0).

In [125]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[125]:
   letter digit
match
0  0  a  1
1  a  2
1  0  b  1
2  0  c  1

In [126]: pd.Series(["a1a2", "b1", "c1"], dtype="string").str.extractall(two_groups)
Out[126]:
   letter digit
match
0  0  a  1
1  a  2
1  0  b  1
2  0  c  1

2.9.7 Testing for strings that match or contain a pattern

You can check whether elements contain a pattern:

In [127]: pattern = r"[\[0-9][a-z]"

In [128]: pd.Series(["1", "2", "3a", "3b", "@3c", "4dx"],
   dtype="string",
   ...: ) .str.contains(pattern)
   ...:

2.9. Working with text data 595
Out[128]:
0    False
1    False
2    True
3    True
4    True
5    True
dtype: boolean

Or whether elements match a pattern:

In [129]: pd.Series(
       ....:     ['1', '2', '3a', '3b', '03c', '4dx'],
       ....:     dtype='string',
       ....: ).str.match(pattern)
       ....: 
Out[129]:
0    False
1    False
2    True
3    True
4    False
5    True
dtype: boolean

New in version 1.1.0.

In [130]: pd.Series(
       ....:     ['1', '2', '3a', '3b', '03c', '4dx'],
       ....:     dtype='string',
       ....: ).str.fullmatch(pattern)
       ....: 
Out[130]:
0    False
1    False
2    True
3    True
4    False
5    False
dtype: boolean

Note: The distinction between match, fullmatch, and contains is strictness: fullmatch tests whether the entire string matches the regular expression; match tests whether there is a match of the regular expression that begins at the first character of the string; and contains tests whether there is a match of the regular expression at any position within the string.

The corresponding functions in the re package for these three match modes are re.fullmatch, re.match, and re.search, respectively.

Methods like match, fullmatch, contains, startswith, and endswith take an extra na argument so missing values can be considered True or False:
In [131]: s4 = pd.Series(
    ....:     ['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'],
    ....:     dtype="string"
    ....: )

In [132]: s4.str.contains("A", na=False)
Out[132]:
0   True
1  False
2  False
3   True
4  False
5  False
6   True
7  False
8  False
dtype: boolean

2.9.8 Creating indicator variables

You can extract dummy variables from string columns. For example if they are separated by a ' | ':

In [133]: s = pd.Series(["a", "a|b", np.nan, "a|c"], dtype="string")

In [134]: s.str.get_dummies(sep="|")
Out[134]:
     a  b  c
0    1  0  0
1    1  1  0
2    0  0  0
3    1  0  1

String Index also supports get_dummies which returns a MultiIndex.

In [135]: idx = pd.Index(["a", "a|b", np.nan, "a|c"])

In [136]: idx.str.get_dummies(sep="|")
Out[136]:
MultiIndex([(1, 0, 0),
            (1, 1, 0),
            (0, 0, 0),
            (1, 0, 1)],
           names=['a', 'b', 'c'])

See also get_dummies().
## 2.9.9 Method summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat()</td>
<td>Concatenate strings</td>
</tr>
<tr>
<td>split()</td>
<td>Split strings on delimiter</td>
</tr>
<tr>
<td>rsplit()</td>
<td>Split strings on delimiter working from the end of the string</td>
</tr>
<tr>
<td>get()</td>
<td>Index into each element (retrieve i-th element)</td>
</tr>
<tr>
<td>join()</td>
<td>Join strings in each element of the Series with passed separator</td>
</tr>
<tr>
<td>get_dummies()</td>
<td>Split strings on the delimiter returning DataFrame of dummy variables</td>
</tr>
<tr>
<td>contains()</td>
<td>Return boolean array if each string contains pattern/regex</td>
</tr>
<tr>
<td>replace()</td>
<td>Replace occurrences of pattern/regex/string with some other string or the return value of a callable given the occurrence</td>
</tr>
<tr>
<td>removeprefix()</td>
<td>Remove prefix from string. i.e. only remove if string starts with prefix.</td>
</tr>
<tr>
<td>removesuffix()</td>
<td>Remove suffix from string, i.e. only remove if string ends with suffix.</td>
</tr>
<tr>
<td>repeat()</td>
<td>Duplicate values (s.str.repeat(3) equivalent to x * 3)</td>
</tr>
<tr>
<td>pad()</td>
<td>Add whitespace to left, right, or both sides of strings</td>
</tr>
<tr>
<td>center()</td>
<td>Equivalent to str.center</td>
</tr>
<tr>
<td>ljust()</td>
<td>Equivalent to str.ljust</td>
</tr>
<tr>
<td>rjust()</td>
<td>Equivalent to str.rjust</td>
</tr>
<tr>
<td>zfill()</td>
<td>Equivalent to str.zfill</td>
</tr>
<tr>
<td>wrap()</td>
<td>Split long strings into lines with length less than a given width</td>
</tr>
<tr>
<td>slice()</td>
<td>Slice each string in the Series</td>
</tr>
<tr>
<td>slice_replace()</td>
<td>Replace slice in each string with passed value</td>
</tr>
<tr>
<td>count()</td>
<td>Count occurrences of pattern</td>
</tr>
<tr>
<td>startswith()</td>
<td>Equivalent to str.startswith(pat) for each element</td>
</tr>
<tr>
<td>endswith()</td>
<td>Equivalent to str.endswith(pat) for each element</td>
</tr>
<tr>
<td>findall()</td>
<td>Compute list of all occurrences of pattern/regex for each string</td>
</tr>
<tr>
<td>match()</td>
<td>Call re.match on each element, returning matched groups as list</td>
</tr>
<tr>
<td>extract()</td>
<td>Call re.search on each element, returning DataFrame with one row for each element and one column for each regex capture group</td>
</tr>
<tr>
<td>extractall()</td>
<td>Call re.findall on each element, returning DataFrame with one row for each regex match and one column for each regex capture group</td>
</tr>
<tr>
<td>len()</td>
<td>Compute string lengths</td>
</tr>
<tr>
<td>strip()</td>
<td>Equivalent to str.strip</td>
</tr>
<tr>
<td>rstrip()</td>
<td>Equivalent to str.rstrip</td>
</tr>
<tr>
<td>lstrip()</td>
<td>Equivalent to str.lstrip</td>
</tr>
<tr>
<td>partition()</td>
<td>Equivalent to str.partition</td>
</tr>
<tr>
<td>rpartition()</td>
<td>Equivalent to str.rpartition</td>
</tr>
<tr>
<td>lower()</td>
<td>Equivalent to str.lower</td>
</tr>
<tr>
<td>casefold()</td>
<td>Equivalent to str.casefold</td>
</tr>
<tr>
<td>upper()</td>
<td>Equivalent to str.upper</td>
</tr>
<tr>
<td>find()</td>
<td>Equivalent to str.find</td>
</tr>
<tr>
<td>rfind()</td>
<td>Equivalent to str.rfind</td>
</tr>
<tr>
<td>index()</td>
<td>Equivalent to str.index</td>
</tr>
<tr>
<td>rindex()</td>
<td>Equivalent to str.rindex</td>
</tr>
<tr>
<td>capitalize()</td>
<td>Equivalent to str.capitalize</td>
</tr>
<tr>
<td>swapcase()</td>
<td>Equivalent to str.swapcase</td>
</tr>
<tr>
<td>normalize()</td>
<td>Return Unicode normal form. Equivalent to unicodedata.normalize</td>
</tr>
<tr>
<td>translate()</td>
<td>Equivalent to str.translate</td>
</tr>
<tr>
<td>isalnum()</td>
<td>Equivalent to str.isalnum</td>
</tr>
<tr>
<td>isalpha()</td>
<td>Equivalent to str.isalpha</td>
</tr>
</tbody>
</table>
2.10 Working with missing data

In this section, we will discuss missing (also referred to as NA) values in pandas.

**Note:** The choice of using NaN internally to denote missing data was largely for simplicity and performance reasons. Starting from pandas 1.0, some optional data types start experimenting with a native NA scalar using a mask-based approach. See [here](#) for more.

See the [cookbook](#) for some advanced strategies.

### 2.10.1 Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python `None` will arise and we wish to also consider that “missing” or “not available” or “NA”.

**Note:** If you want to consider `inf` and `-inf` to be “NA” in computations, you can set `pandas.options.mode.use_inf_as_na = True`.

```
In [1]: df = pd.DataFrame(
      ...:     np.random.randn(5, 3),
      ...:     index=['a', 'c', 'e', 'f', 'h'],
      ...:     columns=['one', 'two', 'three'],
      ...:     )
      ...

In [2]: df['four'] = "bar"

In [3]: df['five'] = df['one'] > 0

In [4]: df
Out[4]:
   one  two  three  four  five
  a  0.469112 -0.282863 -1.509059  bar  True
  c -1.135632  1.212112 -0.173215  bar False
  e  0.119209 -1.044236 -0.861849  bar  True
  f -2.104569 -0.494929  1.071804  bar False
```

(continues on next page)
In[5]: df2 = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])

In[6]: df2

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
<th>four</th>
<th>five</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>bar</td>
<td>True</td>
</tr>
<tr>
<td>b</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>c</td>
<td>-1.135632</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>bar</td>
<td>False</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>e</td>
<td>0.119209</td>
<td>-1.044236</td>
<td>-0.861849</td>
<td>bar</td>
<td>True</td>
</tr>
<tr>
<td>f</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
<td>bar</td>
<td>False</td>
</tr>
<tr>
<td>g</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>h</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>bar</td>
<td>True</td>
</tr>
</tbody>
</table>

To make detecting missing values easier (and across different array dtypes), pandas provides the `isna()` and `notna()` functions, which are also methods on Series and DataFrame objects:

In[7]: df2["one"]

Out[7]:

<table>
<thead>
<tr>
<th></th>
<th>one</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.469112</td>
</tr>
<tr>
<td>b</td>
<td>NaN</td>
</tr>
<tr>
<td>c</td>
<td>-1.135632</td>
</tr>
<tr>
<td>d</td>
<td>NaN</td>
</tr>
<tr>
<td>e</td>
<td>0.119209</td>
</tr>
<tr>
<td>f</td>
<td>-2.104569</td>
</tr>
<tr>
<td>g</td>
<td>NaN</td>
</tr>
<tr>
<td>h</td>
<td>0.721555</td>
</tr>
</tbody>
</table>

Name: one, dtype: float64

In[8]: pd.isna(df2["one"])

Out[8]:

<table>
<thead>
<tr>
<th></th>
<th>isna()</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>False</td>
</tr>
<tr>
<td>b</td>
<td>True</td>
</tr>
<tr>
<td>c</td>
<td>False</td>
</tr>
<tr>
<td>d</td>
<td>True</td>
</tr>
<tr>
<td>e</td>
<td>False</td>
</tr>
<tr>
<td>f</td>
<td>False</td>
</tr>
<tr>
<td>g</td>
<td>True</td>
</tr>
<tr>
<td>h</td>
<td>False</td>
</tr>
</tbody>
</table>

Name: one, dtype: bool

In[9]: df2["four"].notna()

Out[9]:

<table>
<thead>
<tr>
<th></th>
<th>notna()</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>True</td>
</tr>
<tr>
<td>b</td>
<td>False</td>
</tr>
<tr>
<td>c</td>
<td>True</td>
</tr>
<tr>
<td>d</td>
<td>False</td>
</tr>
<tr>
<td>e</td>
<td>True</td>
</tr>
<tr>
<td>f</td>
<td>True</td>
</tr>
<tr>
<td>g</td>
<td>False</td>
</tr>
</tbody>
</table>

(continues on next page)
h    True
Name: four, dtype: bool

In [10]: df2.isna()
Out[10]:
      one  two  three  four  five
a  False  False  False  False  False
b    True   True   True   True
  True   True   True   True
  True   True   True   True
  True   True   True   True
  True   True   True   True
  True   True   True   True
  True   True   True   True
      False  False  False  False  False

Warning:  One has to be mindful that in Python (and NumPy), the nan’s don’t compare equal, but None’s do. Note that pandas/NumPy uses the fact that np.nan != np.nan, and treats None like np.nan.

In [11]: None == None  # noqa: E711
Out[11]: True

In [12]: np.nan == np.nan
Out[12]: False

So as compared to above, a scalar equality comparison versus a None/np.nan doesn’t provide useful information.

In [13]: df2["one"] == np.nan
Out[13]:
a    False
b    False
c    False
d    False
e    False
f    False
g    False
h    False
Name: one, dtype: bool

Integer dtypes and missing data

Because NaN is a float, a column of integers with even one missing values is cast to floating-point dtype (see Support for integer NA for more). pandas provides a nullable integer array, which can be used by explicitly requesting the dtype:

In [14]: pd.Series([1, 2, np.nan, 4], dtype=pd.Int64Dtype())
Out[14]:
      0  1
      1  2
      2 <NA>
      3  4
dtype: Int64
Alternatively, the string alias `dtype='Int64'` (note the capital “I”) can be used.

See Nullable integer data type for more.

**Datetimes**

For datetime64[ns] types, `NaT` represents missing values. This is a pseudo-native sentinel value that can be represented by NumPy in a singular dtype (datetime64[ns]). pandas objects provide compatibility between `NaT` and `NaN`.

```python
In [15]: df2 = df.copy()

In [16]: df2["timestamp"] = pd.Timestamp("20120101")

In [17]: df2
```

```
one two three four five timestamp
a 0.469112 -0.282863 -1.509059 bar True 2012-01-01
c -1.135632 1.212112 -0.173215 bar False 2012-01-01
e 0.119209 -1.044236 -0.861849 bar True 2012-01-01
f -2.104569 -0.494929 1.071804 bar False 2012-01-01
h 0.721555 -0.706771 -1.039575 bar True 2012-01-01
```

```python
In [18]: df2.loc["a", "c", "h"], ["one", "timestamp"] = np.nan

In [19]: df2
```

```
one two three four five timestamp
a NaN -0.282863 -1.509059 bar True NaT
c NaN 1.212112 -0.173215 bar False NaT
e NaN -1.044236 -0.861849 bar True 2012-01-01
f -2.104569 -0.494929 1.071804 bar False 2012-01-01
h NaN -0.706771 -1.039575 bar True NaT
```

```python
In [20]: df2.dtypes.value_counts()
```

```
float64 3
object 1
bool 1
datetime64[ns] 1
dtype: int64
```

### 2.10.2 Inserting missing data

You can insert missing values by simply assigning to containers. The actual missing value used will be chosen based on the dtype.

For example, numeric containers will always use `NaN` regardless of the missing value type chosen:

```python
In [21]: s = pd.Series([1, 2, 3])

In [22]: s.loc[0] = None
```
Likewise, datetime containers will always use NaT.

For object containers, pandas will use the value given:

```python
In [24]: s = pd.Series(["a", "b", "c"])
In [25]: s.loc[0] = None
In [26]: s.loc[1] = np.nan
In [27]: s
```

```
Out[27]:
0   None
1   NaN
2    c
dtype: object
```

### 2.10.3 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```python
In [28]: a
```

```
Out[28]:
     one  two
  a     NaN  -0.282863
  c     NaN   1.212112
  e  0.119209  -1.044236
  f -2.104569   -0.494929
  h -2.104569  -0.706771
```

```python
In [29]: b
```

```
Out[29]:
     one  two  three
  a     NaN  NaN  -1.509059
  c     NaN  NaN  -0.173215
  e  0.119209 -1.044236  -0.861849
  f -2.104569 -0.494929   1.071804
  h     NaN  NaN  -1.039575
```

```python
In [30]: a + b
```

```
Out[30]:
     one  three  two
  a     NaN   NaN   NaN
  c     NaN   NaN   2.424224
```
The descriptive statistics and computational methods discussed in the data structure overview (and listed here and here) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero.
- If the data are all NA, the result will be 0.
- Cumulative methods like `cumsum()` and `cumprod()` ignore NA values by default, but preserve them in the resulting arrays. To override this behaviour and include NA values, use `skipna=False`.

```
In [31]: df
Out[31]:
     one   two   three
a  NaN -0.282863 -1.509059
c  NaN  1.212112  -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h  NaN  -0.706771 -1.039575

In [32]: df["one"].sum()
Out[32]: -1.9853605075978744

In [33]: df.mean(1)
Out[33]:
a -0.895961
c  0.519449
e -0.595625
f -0.509232
h -0.873173
dtype: float64

In [34]: df.cumsum()
Out[34]:
     one   two   three
a  NaN -0.282863 -1.509059
c  NaN  1.212112  -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h  NaN  -0.706771 -1.039575

In [35]: df.cumsum(skipna=False)
Out[35]:
     one   two   three
a  NaN -0.282863 -1.509059
c  NaN  0.929249  -1.682273
e  0.119209 -0.114987 -2.544122
f -1.985361 -0.609917  -1.472318
h  NaN -1.316688 -2.511893
```
2.10.4 Sum/prod of empties/nans

**Warning:** This behavior is now standard as of v0.22.0 and is consistent with the default in numpy; previously sum/prod of all-NA or empty Series/DataFrames would return NaN. See v0.22.0 whatsnw for more.

The sum of an empty or all-NA Series or column of a DataFrame is 0.

```
In [36]: pd.Series([np.nan]).sum()
Out[36]: 0.0
```

```
In [37]: pd.Series([], dtype="float64").sum()
Out[37]: 0.0
```

The product of an empty or all-NA Series or column of a DataFrame is 1.

```
In [38]: pd.Series([np.nan]).prod()
Out[38]: 1.0
```

```
In [39]: pd.Series([], dtype=“float64”).prod()
Out[39]: 1.0
```

2.10.5 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example:

```
In [40]: df
Out[40]:
   one   two   three
a NaN -0.282863 -1.509059
c NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h NaN -0.706771 -1.039575
```

```
In [41]: df.groupby("one").mean()
Out[41]:
   two    three
one
-2.104569 -0.494929  1.071804
  0.119209 -1.044236 -0.861849
```

See the groupby section *here* for more information.
2.10.6 Filling missing values: fillna

`fillna()` can “fill in” NA values with non-NA data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [42]: df2
Out[42]:
     one    two    three    four five timestamp
   a NaN -0.282863 -1.509059   bar   True    NaT
   c NaN  1.212112 -0.173215  bar   False    NaT
  e 0.119209 -1.044236 -0.861849   bar    True 2012-01-01
  f 2.104569  1.071804  1.120429   bar   False    NaT
  h NaN -0.706771 -1.039575   bar    True 2012-01-01

In [43]: df2.fillna(0)
Out[43]:
     one    two    three    four five    timestamp
   a 0.000000 -0.282863 -1.509059   bar    True      0
   c 0.000000  1.212112 -0.173215  bar   False      0
  e 0.119209 -1.044236 -0.861849   bar    True 2012-01-01 00:00:00
  f 2.104569  1.071804  1.120429   bar   False 2012-01-01 00:00:00
  h 0.000000 -0.706771 -1.039575   bar    True      0
```

Fill gaps forward or backward

Using the same filling arguments as `reindexing`, we can propagate non-NA values forward or backward:

```
In [44]: df2["one"].fillna("missing")
Out[44]:
    a    missing
   c    missing
  e 0.119209
  f 2.104569
  h    missing
Name: one, dtype: object
```

(continues on next page)
Limit the amount of filling

If we only want consecutive gaps filled up to a certain number of data points, we can use the `limit` keyword:

```{}
In [47]: df
Out[47]:
     one  two  three
a  NaN  0.282863  -1.509059
c  NaN  1.212112  -0.173215
e  NaN  NaN      NaN
f  NaN  NaN      NaN
h  NaN  0.706771  -1.039575
```

```{}
In [48]: df.fillna(method="pad", limit=1)
Out[48]:
     one  two  three
a  NaN  0.282863  -1.509059
c  NaN  1.212112  -0.173215
e  NaN  1.212112  -0.173215
f  NaN  NaN      NaN
h  NaN  0.706771  -1.039575
```

To remind you, these are the available filling methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad / ffill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
</tbody>
</table>

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

`ffill()` is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

### 2.10.7 Filling with a PandasObject

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```{}
In [49]: dff = pd.DataFrame(np.random.randn(10, 3), columns=list("ABC"))
In [50]: dff.iloc[3:5, 0] = np.nan
In [51]: dff.iloc[4:6, 1] = np.nan
In [52]: dff.iloc[5:8, 2] = np.nan
In [53]: dff
```

(continues on next page)
Out[53]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.27186</td>
<td>-0.42497</td>
<td>0.56702</td>
</tr>
<tr>
<td>1</td>
<td>0.27623</td>
<td>-1.08740</td>
<td>-0.67369</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>NaN</td>
<td>-1.157892</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>

In [54]: dff.fillna(dff.mean())

Out[54]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.27186</td>
<td>-0.42497</td>
<td>0.56702</td>
</tr>
<tr>
<td>1</td>
<td>0.27623</td>
<td>-1.08740</td>
<td>-0.67369</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>-0.140857</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>-0.140857</td>
<td>-0.401419</td>
<td>-0.293543</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>-0.401419</td>
<td>-0.293543</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>-0.293543</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>-0.293543</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>

In [55]: dff.fillna(dff.mean()["B":"C"])

Out[55]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.27186</td>
<td>-0.42497</td>
<td>0.56702</td>
</tr>
<tr>
<td>1</td>
<td>0.27623</td>
<td>-1.08740</td>
<td>-0.67369</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>NaN</td>
<td>-1.157892</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>

Same result as above, but is aligning the ‘fill’ value which is a Series in this case.

In [56]: dff.where(pd.notna(dff), dff.mean(), axis="columns")

Out[56]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.27186</td>
<td>-0.42497</td>
<td>0.56702</td>
</tr>
<tr>
<td>1</td>
<td>0.27623</td>
<td>-1.08740</td>
<td>-0.67369</td>
</tr>
<tr>
<td>2</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>3</td>
<td>-0.140857</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>4</td>
<td>-0.140857</td>
<td>-0.401419</td>
<td>-0.293543</td>
</tr>
<tr>
<td>5</td>
<td>-1.344312</td>
<td>-0.401419</td>
<td>-0.293543</td>
</tr>
<tr>
<td>6</td>
<td>-0.109050</td>
<td>1.643563</td>
<td>-0.293543</td>
</tr>
<tr>
<td>7</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>-0.293543</td>
</tr>
<tr>
<td>8</td>
<td>-0.968914</td>
<td>-1.294524</td>
<td>0.413738</td>
</tr>
<tr>
<td>9</td>
<td>0.276662</td>
<td>-0.472035</td>
<td>-0.013960</td>
</tr>
</tbody>
</table>
2.10.8 Dropping axis labels with missing data: dropna

You may wish to simply exclude labels from a data set which refer to missing data. To do this, use `dropna()`:

```
In [57]: df
Out[57]:
one  two  three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN   0.000000   0.000000
f  NaN   0.000000   0.000000
h  NaN -0.706771 -1.039575

In [58]: df.dropna(axis=0)
Out[58]:
Empty DataFrame
Columns: [one, two, three]
Index: []

In [59]: df.dropna(axis=1)
Out[59]:
   two  three
  a -0.282863 -1.509059
c  1.212112 -0.173215
e   0.000000   0.000000
f   0.000000   0.000000
h -0.706771 -1.039575

In [60]: df["one"].dropna()
Out[60]: Series([], Name: one, dtype: float64)
```

An equivalent `dropna()` is available for Series. DataFrame.dropna has considerably more options than Series.dropna, which can be examined in the API.

2.10.9 Interpolation

Both Series and DataFrame objects have `interpolate()` that, by default, performs linear interpolation at missing data points.

```
In [61]: ts
Out[61]:
2000-01-31  0.469112
2000-02-29   NaN
2000-03-31   NaN
2000-04-28   NaN
```

(continues on next page)
2000-05-31   NaN
...  
2007-12-31  -6.950267
2008-01-31  -7.904475
2008-02-29  -6.441779
2008-03-31  -8.184940
2008-04-30  -9.011531
Freq: BM, Length: 100, dtype: float64

In [62]: ts.count()
Out[62]: 66

In [63]: ts.plot()
Out[63]: <AxesSubplot:>

In [64]: ts.interpolate()
Out[64]:
2000-01-31   0.469112
2000-02-29   0.434469
2000-03-31   0.399826
2000-04-28   0.365184
2000-05-31   0.330541
Index-aware interpolation is available via the method keyword:

```
In [67]: ts2
Out[67]:
2000-01-31  0.469112
2000-02-29  NaN
2002-07-31 -5.785037
2005-01-31  NaN
```

(continues on next page)
In [68]: ts2.interpolate()
Out[68]:
2000-01-31  0.469112
2000-02-29 -2.657962
2002-07-31 -5.785037
2005-01-31 -7.190866
2008-04-30 -9.011531
dtype: float64

In [69]: ts2.interpolate(method="time")
Out[69]:
2000-01-31  0.469112
2000-02-29  0.270241
2002-07-31 -5.785037
2005-01-31 -7.190866
2008-04-30 -9.011531
dtype: float64

For a floating-point index, use method='values':

In [70]: ser
Out[70]:
0.0  0.0
1.0  NaN
10.0 10.0
dtype: float64

In [71]: ser.interpolate()
Out[71]:
0.0  0.0
1.0  5.0
10.0 10.0
dtype: float64

In [72]: ser.interpolate(method="values")
Out[72]:
0.0  0.0
1.0  1.0
10.0 10.0
dtype: float64

You can also interpolate with a DataFrame:

In [73]: df = pd.DataFrame(
.....:   {  
.....:     "A": [1, 2.1, np.nan, 4.7, 5.6, 6.8],  
.....:     "B": [0.25, np.nan, np.nan, 4, 12.2, 14.4],  
.....:   },  
.....: )
(continues on next page)
The `method` argument gives access to fancier interpolation methods. If you have scipy installed, you can pass the name of a 1-d interpolation routine to `method`. You’ll want to consult the full scipy interpolation documentation and reference guide for details. The appropriate interpolation method will depend on the type of data you are working with.

- If you are dealing with a time series that is growing at an increasing rate, `method='quadratic'` may be appropriate.
- If you have values approximating a cumulative distribution function, then `method='pchip'` should work well.
- To fill missing values with goal of smooth plotting, consider `method='akima'`.

**Warning:** These methods require scipy.
When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [98]: df.interpolate(method="spline", order=2)
Out[98]:
   A     B
0 1.000000  0.250000
1 2.100000 -0.428598
2 3.404545  1.206900
3 4.700000  4.000000
4 5.600000 12.200000
5 6.800000 14.400000
```

Compare several methods:

```
In [101]: np.random.seed(2)

In [102]: ser = pd.Series(np.arange(1, 10.1, 0.25) ** 2 + np.random.randn(37))

In [103]: missing = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])

In [104]: ser[missing] = np.nan

In [105]: methods = ["linear", "quadratic", "cubic"]

In [106]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})

In [107]: df.plot()
Out[107]: <AxesSubplot:>
```
Another use case is interpolation at new values. Suppose you have 100 observations from some distribution. And let’s suppose that you’re particularly interested in what’s happening around the middle. You can mix pandas’ reindex and interpolate methods to interpolate at the new values.

```python
In [88]: ser = pd.Series(np.sort(np.random.uniform(size=100)))
# interpolate at new_index
In [89]: new_index = ser.index.union(pd.Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75]))
In [90]: interp_s = ser.reindex(new_index).interpolate(method="pchip")
In [91]: interp_s[49:51]
Out[91]:
   49.00    0.471410
   49.25    0.476841
   49.50    0.481780
   49.75    0.485998
   50.00    0.489266
   50.25    0.491814
   50.50    0.493995
   50.75    0.495763
   51.00    0.497074
dtype: float64
```

2.10. Working with missing data
Interpolation limits

Like other pandas fill methods, `interpolate()` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive NaN values filled since the last valid observation:

```
In [92]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13, np.nan, np.nan])

In [93]: ser
Out[93]:
0    NaN
1    NaN
2    5.0
3    NaN
4    NaN
5    NaN
6    13.0
7    NaN
8    NaN
dtype: float64

# fill all consecutive values in a forward direction
In [94]: ser.interpolate()
Out[94]:
0    NaN
1    NaN
2    5.0
3    7.0
4    9.0
5    11.0
6    13.0
7    13.0
8    13.0
dtype: float64

# fill one consecutive value in a forward direction
In [95]: ser.interpolate(limit=1)
Out[95]:
0    NaN
1    NaN
2    5.0
3    7.0
4    NaN
5    NaN
6    13.0
7    13.0
8    NaN
dtype: float64
```

By default, NaN values are filled in a forward direction. Use `limit_direction` parameter to fill backward or from both directions.

```
# fill one consecutive value backwards
In [96]: ser.interpolate(limit=1, limit_direction="backward")
Out[96]:
```

(continues on next page)
# fill one consecutive value in both directions
In [97]: ser.interpolate(limit=1, limit_direction="both")

Out[97]:
0  NaN
1  5.0
2  5.0
3  NaN
4  NaN
5 11.0
6 13.0
7  NaN
8  NaN
dtype: float64

# fill all consecutive values in both directions
In [98]: ser.interpolate(limit_direction="both")

Out[98]:
0  5.0
1  5.0
2  5.0
3  7.0
4  9.0
5 11.0
6 13.0
7 13.0
8 13.0
dtype: float64

By default, NaN values are filled whether they are inside (surrounded by) existing valid values, or outside existing valid values. The limit_area parameter restricts filling to either inside or outside values.

# fill one consecutive inside value in both directions
In [99]: ser.interpolate(limit_direction="both", limit_area="inside", limit=1)

Out[99]:
0  NaN
1  NaN
2  5.0
3  7.0
4  NaN
5 11.0
6 13.0

(continues on next page)
2.10.10 Replacing generic values

Often times we want to replace arbitrary values with other values.

*replace()* in Series and *replace()* in DataFrame provides an efficient yet flexible way to perform such replacements.

For a Series, you can replace a single value or a list of values by another value:

```python
In [102]: ser = pd.Series([0.0, 1.0, 2.0, 3.0, 4.0])
In [103]: ser.replace(0, 5)
Out[103]:
0   5.0
1   1.0
2   2.0
3   3.0
4   4.0
 dtype: float64
```

You can replace a list of values by a list of other values:

```python
In [102]: ser = pd.Series([0.0, 1.0, 2.0, 3.0, 4.0])
In [103]: ser.replace({0: 5, 1: 1, 2: 2, 3: 3, 4: 4})
Out[103]:
0   5.0
1   1.0
2   2.0
3   3.0
4   4.0
 dtype: float64
```
In [104]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[104]:
0   4.0
1   3.0
2   2.0
3   1.0
4   0.0
dtype: float64

You can also specify a mapping dict:

In [105]: ser.replace({0: 10, 1: 100})
Out[105]:
0   10.0
1  100.0
2   2.0
3   3.0
4   4.0
dtype: float64

For a DataFrame, you can specify individual values by column:

In [106]: df = pd.DataFrame({"a": [0, 1, 2, 3, 4], "b": [5, 6, 7, 8, 9]})

In [107]: df.replace({"a": 0, "b": 5}, 100)
Out[107]:
    a  b
0  100  100
1   1   6
2   2   7
3   3   8
4   4   9

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

In [108]: ser.replace([1, 2, 3], method="pad")
Out[108]:
0   0.0
1   0.0
2   0.0
3   0.0
4   4.0
dtype: float64
2.10.11 String/regular expression replacement

Note: Python strings prefixed with the `r` character such as `r'hello world'` are so-called “raw” strings. They have different semantics regarding backslashes than strings without this prefix. Backslashes in raw strings will be interpreted as an escaped backslash, e.g., `r'\' == '\\'. You should read about them if this is unclear.

Replace the ‘.’ with `NaN` (str -> str):

```
In [109]: d = {"a": list(range(4)), "b": list("ab.."), "c": ["a", "b", np.nan, "d"]}
In [110]: df = pd.DataFrame(d)
In [111]: df.replace(".", np.nan)
Out[111]:
a  b  c
0  0  a  a
1  1  b  b
2  NaN  NaN
3  NaN  d
```

Now do it with a regular expression that removes surrounding whitespace (regex -> regex):

```
In [112]: df.replace(r"\s*.\s*", np.nan, regex=True)
Out[112]:
a  b  c
0  0  a  a
1  1  b  b
2  NaN  NaN
3  NaN  d
```

Replace a few different values (list -> list):

```
In [113]: df.replace(["a", "."], ["b", np.nan])
Out[113]:
a  b  c
0  0  b  b
1  1  b  b
2  NaN  NaN
3  NaN  d
```

list of regex -> list of regex:

```
In [114]: df.replace([r"\."], ["dot", r"\1stuff"], regex=True)
Out[114]:
a  b  c
0  0  astuff  astuff
1  1  b  b
2  dot  NaN
3  dot  d
```

Only search in column 'b' (dict -> dict):

```
```
In [115]: df.replace({'b': ' '}, {'b': np.nan})
Out[115]:
    a  b  c
0  0  a  a
1  1  b  b
2  NaN NaN
3  NaN  d

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict):

In [116]: df.replace({'b': r'\s*\s'}, {'b': np.nan}, regex=True)
Out[116]:
    a  b  c
0  0  a  a
1  1  b  b
2  NaN NaN
3  NaN  d

You can pass nested dictionaries of regular expressions that use regex=True:

In [117]: df.replace({'b': {'b': r''}}, regex=True)
Out[117]:
    a  b  c
0  0  a  a
1  1  b
2  .  NaN
3  .  d

Alternatively, you can pass the nested dictionary like so:

In [118]: df.replace(regex={'b': {r'\s*\s': np.nan}})
Out[118]:
    a  b  c
0  0  a  a
1  1  b
2  NaN NaN
3  NaN  d

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well.

In [119]: df.replace({'b': r'\s*(\.)\s'}, {'b': r'^ly'}, regex=True)
Out[119]:
    a  b  c
0  0  a  a
1  1  b
2  .ly NaN
3  .ly  d

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex).

In [120]: df.replace([r'\s*\s', r'a|b'], np.nan, regex=True)
Out[120]:
(continues on next page)
All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be:

```
In [121]: df.replace(regex=[r"\s*\./\s*", r"a|b"], value=np.nan)
```

```
Out[121]:
    a   b   c
0  NaN  NaN
1  NaN  NaN
2  NaN  NaN
3  NaN   d
```

This can be convenient if you do not want to pass `regex=True` every time you want to use a regular expression.

**Note:** Anywhere in the above `replace` examples that you see a regular expression a compiled regular expression is valid as well.

### 2.10.12 Numeric replacement

`replace()` is similar to `fillna()`.

```
In [122]: df = pd.DataFrame(np.random.randn(10, 2))
In [123]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5
In [124]: df.replace(1.5, np.nan)
```

```
Out[124]:
       0   1
0 -0.844214 -1.021415
1  0.432396  0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4   NaN   NaN
5   NaN   NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9   NaN   NaN
```

Replacing more than one value is possible by passing a list.

```
In [125]: df00 = df.iloc[0, 0]
In [126]: df.replace([1.5, df00], [np.nan, "a"])
```

(continues on next page)
Out[126]:

```
     0     1
0  a -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4 NaN  NaN
5 NaN  NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9 NaN  NaN
```

In [127]: df[1].dtype
Out[127]: dtype('float64')

You can also operate on the DataFrame in place:

In [128]: df.replace(1.5, np.nan, inplace=True)

### Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we’ve established some “casting rules”. When a reindexing operation introduces missing data, the Series will be cast according to the rules introduced in the table below.

<table>
<thead>
<tr>
<th>data type</th>
<th>Cast to</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>float</td>
</tr>
<tr>
<td>boolean</td>
<td>object</td>
</tr>
<tr>
<td>float</td>
<td>no cast</td>
</tr>
<tr>
<td>object</td>
<td>no cast</td>
</tr>
</tbody>
</table>

For example:

In [129]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [130]: s > 0
Out[130]:

```
0   True
1   True
2   True
3   True
4   True
5   True
dtype: bool
```

In [131]: (s > 0).dtype
Out[131]: dtype('bool')

In [132]: crit = (s > 0).reindex(list(range(8)))

(continues on next page)
Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

```
In [135]: reindexed = s.reindex(list(range(8))).fillna(0)
In [136]: reindexed[crit]
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
Input In [136], in <cell line: 1>()
----> 1 reindexed[crit]

File /pandas/pandas/core/series.py:979, in Series.__getitem__(self, key)
    976     if is_iterator(key):
    977         key = list(key)
--> 979     if com.is_bool_indexer(key):
    980         key = check_bool_indexer(self.index, key)
    981         key = np.asarray(key, dtype=bool)

File /pandas/pandas/core/common.py:144, in is_bool_indexer(key)
   140     na_msg = "Cannot mask with non-boolean array containing NA / NaN values"
   141     if lib.infer_dtype(key) == "boolean" and isna(key).any():
   142         # Don't raise on e.g. ["A", "B", np.nan], see
   143         # test_loc_getitem_list_of_labels_categoricalindex_with_na
---> 144         raise ValueError(na_msg)
   145     return False
   146 return True

ValueError: Cannot mask with non-boolean array containing NA / NaN values
```

However, these can be filled in using `fillna()` and it will work fine:

```
In [137]: reindexed[crit.fillna(False)]
Out[137]:
0   0.126504
2   0.696198
4   0.697416
```
6 0.601516
7 0.003659
dtype: float64

In [138]: reindexed[crit.fillna(True)]
Out[138]:
0 0.126504
1 0.000000
2 0.696198
3 0.000000
4 0.697416
5 0.000000
6 0.601516
7 0.003659
dtype: float64

pandas provides a nullable integer dtype, but you must explicitly request it when creating the series or column. Notice that we use a capital “I” in the dtype=”Int64”.

In [139]: s = pd.Series([0, 1, np.nan, 3, 4], dtype="Int64")

In [140]: s
Out[140]:
0 0
1 1
2 <NA>
3 3
4 4
dtype: Int64

See Nullable integer data type for more.

2.10.13 Experimental NA scalar to denote missing values

**Warning:** Experimental: the behaviour of pd.NA can still change without warning.

New in version 1.0.0.

Starting from pandas 1.0, an experimental pd.NA value (singleton) is available to represent scalar missing values. At this moment, it is used in the nullable integer, boolean and dedicated string data types as the missing value indicator.

The goal of pd.NA is provide a “missing” indicator that can be used consistently across data types (instead of np.nan, None or pd.NaT depending on the data type).

For example, when having missing values in a Series with the nullable integer dtype, it will use pd.NA:

In [141]: s = pd.Series([1, 2, None], dtype="Int64")

In [142]: s
Out[142]:
0 1
(continues on next page)
Currently, pandas does not yet use those data types by default (when creating a DataFrame or Series, or when reading in data), so you need to specify the dtype explicitly. An easy way to convert to those dtypes is explained [here](#).

## Propagation in arithmetic and comparison operations

In general, missing values *propagate* in operations involving `pd.NA`. When one of the operands is unknown, the outcome of the operation is also unknown.

For example, `pd.NA` propagates in arithmetic operations, similarly to `np.nan`:

```python
In [145]: pd.NA + 1
Out[145]: <NA>
```

```python
In [146]: "a" * pd.NA
Out[146]: <NA>
```

There are a few special cases when the result is known, even when one of the operands is `NA`:

```python
In [147]: pd.NA ** 0
Out[147]: 1
```

```python
In [148]: 1 ** pd.NA
Out[148]: 1
```

In equality and comparison operations, `pd.NA` also propagates. This deviates from the behaviour of `np.nan`, where comparisons with `np.nan` always return `False`:

```python
In [149]: pd.NA == 1
Out[149]: <NA>
```

```python
In [150]: pd.NA == pd.NA
Out[150]: <NA>
```

```python
In [151]: pd.NA < 2.5
Out[151]: <NA>
```

To check if a value is equal to `pd.NA`, the `isna()` function can be used:

```python
In [152]: pd.isna(pd.NA)
Out[152]: True
```

An exception on this basic propagation rule are *reductions* (such as the mean or the minimum), where pandas defaults to skipping missing values. See [above](#) for more.
Logical operations

For logical operations, `pd.NA` follows the rules of the three-valued logic (or Kleene logic, similarly to R, SQL and Julia). This logic means to only propagate missing values when it is logically required.

For example, for the logical “or” operation (`|`), if one of the operands is `True`, we already know the result will be `True`, regardless of the other value (so regardless the missing value would be `True` or `False`). In this case, `pd.NA` does not propagate:

```
In [153]: True | False
Out[153]: True

In [154]: True | pd.NA
Out[154]: True

In [155]: pd.NA | True
Out[155]: True
```

On the other hand, if one of the operands is `False`, the result depends on the value of the other operand. Therefore, in this case `pd.NA` propagates:

```
In [156]: False | True
Out[156]: True

In [157]: False | False
Out[157]: False

In [158]: False | pd.NA
Out[158]: <NA>
```

The behaviour of the logical “and” operation (`&`) can be derived using similar logic (where now `pd.NA` will not propagate if one of the operands is already `False`):

```
In [159]: False & True
Out[159]: False

In [160]: False & False
Out[160]: False

In [161]: False & pd.NA
Out[161]: False

In [162]: True & True
Out[162]: True

In [163]: True & False
Out[163]: False

In [164]: True & pd.NA
Out[164]: <NA>
```
**NA in a boolean context**

Since the actual value of an NA is unknown, it is ambiguous to convert NA to a boolean value. The following raises an error:

```python
In [165]: bool(pd.NA)
---------------------------------------------------------------------------
TypeError                          Traceback (most recent call last)
Input In [165], in <cell line: 1>()
----> 1 bool(pd.NA)

File /pandas/pandas/_libs/missing.pyx:382, in pandas._libs.missing.NAType.__bool__()
TypeError: boolean value of NA is ambiguous
```

This also means that `pd.NA` cannot be used in a context where it is evaluated to a boolean, such as `if condition:` ... where `condition` can potentially be `pd.NA`. In such cases, `isna()` can be used to check for `pd.NA` or condition being `pd.NA` can be avoided, for example by filling missing values beforehand.

A similar situation occurs when using Series or DataFrame objects in if statements, see *Using if/truth statements with pandas*.

**NumPy ufuncs**

`pandas.NA` implements NumPy’s `__array_ufunc__` protocol. Most ufuncs work with NA, and generally return NA:

```python
In [166]: np.log(pd.NA)
Out[166]: <NA>

In [167]: np.add(pd.NA, 1)
Out[167]: <NA>
```

**Warning:** Currently, ufuncs involving an ndarray and NA will return an object-dtype filled with NA values.

```python
In [168]: a = np.array([1, 2, 3])

In [169]: np.greater(a, pd.NA)
Out[169]: array([<NA>, <NA>, <NA>], dtype=object)
```

The return type here may change to return a different array type in the future.

See *DataFrame interoperability with NumPy functions* for more on ufuncs.
Conversion

If you have a DataFrame or Series using traditional types that have missing data represented using np.nan, there are convenience methods convert_dtypes() in Series and convert_dtypes() in DataFrame that can convert data to use the newer dtypes for integers, strings and booleans listed here. This is especially helpful after reading in data sets when letting the readers such as read_csv() and read_excel() infer default dtypes.

In this example, while the dtypes of all columns are changed, we show the results for the first 10 columns.

```
In [170]: bb = pd.read_csv("data/baseball.csv", index_col="id")

In [171]: bb[bb.columns[:10]].dtypes

Out[171]:
player   object
year     int64
stint    int64
team     object
lg       object
g        int64
ab       int64
r         int64
h         int64
X2b       int64
dtype: object
```

```
In [172]: bbn = bb.convert_dtypes()

In [173]: bbn[bbn.columns[:10]].dtypes

Out[173]:
player   string
year     Int64
stint    Int64
team     string
lg       string
g        Int64
ab       Int64
r         Int64
h         Int64
X2b       Int64
dtype: object
```

2.11 Duplicate Labels

Index objects are not required to be unique; you can have duplicate row or column labels. This may be a bit confusing at first. If you’re familiar with SQL, you know that row labels are similar to a primary key on a table, and you would never want duplicates in a SQL table. But one of pandas’ roles is to clean messy, real-world data before it goes to some downstream system. And real-world data has duplicates, even in fields that are supposed to be unique.

This section describes how duplicate labels change the behavior of certain operations, and how prevent duplicates from arising during operations, or to detect them if they do.
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [1]: import pandas as pd
In [2]: import numpy as np

2.11.1 Consequences of Duplicate Labels
Some pandas methods (Series.reindex() for example) just don’t work with duplicates present. The output can’t be
determined, and so pandas raises.
In [3]: s1 = pd.Series([0, 1, 2], index=["a", "b", "b"])
In [4]: s1.reindex(["a", "b", "c"])
--------------------------------------------------------------------------ValueError
Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 s1.reindex(["a", "b", "c"])
File /pandas/pandas/core/series.py:4672, in Series.reindex(self, *args, **kwargs)
4668
raise TypeError(
4669
"'index' passed as both positional and keyword argument"
4670
)
4671
kwargs.update({"index": index})
-> 4672 return super().reindex(**kwargs)
File /pandas/pandas/core/generic.py:4966, in NDFrame.reindex(self, *args, **kwargs)
4963
return self._reindex_multi(axes, copy, fill_value)
4965 # perform the reindex on the axes
-> 4966 return self._reindex_axes(
4967
axes, level, limit, tolerance, method, fill_value, copy
4968 ).__finalize__(self, method="reindex")
File /pandas/pandas/core/generic.py:4986, in NDFrame._reindex_axes(self, axes, level,␣
˓→limit, tolerance, method, fill_value, copy)
4981 new_index, indexer = ax.reindex(
4982
labels, level=level, limit=limit, tolerance=tolerance, method=method
4983 )
4985 axis = self._get_axis_number(a)
-> 4986 obj = obj._reindex_with_indexers(
4987
{axis: [new_index, indexer]},
4988
fill_value=fill_value,
4989
copy=copy,
4990
allow_dups=False,
4991 )
4992 # If we've made a copy once, no need to make another one
4993 copy = False
File /pandas/pandas/core/generic.py:5032, in NDFrame._reindex_with_indexers(self,␣
˓→reindexers, fill_value, copy, allow_dups)
5029
indexer = ensure_platform_int(indexer)
5031 # TODO: speed up on homogeneous DataFrame objects (see _reindex_multi)
-> 5032 new_data = new_data.reindex_indexer(
(continues on next page)

630

Chapter 2. User Guide


Other methods, like indexing, can give very surprising results. Typically indexing with a scalar will reduce dimensionality. Slicing a DataFrame with a scalar will return a Series. Slicing a Series with a scalar will return a scalar. But with duplicates, this isn’t the case.

```
in [5]: df1 = pd.DataFrame([[0, 1, 2], [3, 4, 5]], columns=["A", "A", "B"])

In [6]: df1
Out[6]:
   A  A  B
0  0  1  2
1  3  4  5
```

We have duplicates in the columns. If we slice 'B', we get back a Series

```
in [7]: df1["B"]  # a series
Out[7]:
   0  2
   1  5
Name: B, dtype: int64
```

But slicing 'A' returns a DataFrame

```
in [8]: df1["A"]  # a DataFrame
Out[8]:
   A  A
```

(continues on next page)
This applies to row labels as well:

```python
In [9]: df2 = pd.DataFrame({"A": [0, 1, 2]}, index=["a", "a", "b"])
Out[9]:
   A
a 0
a 1
b 2
```

```python
In [10]: df2.loc["b", "A"]  # a scalar
Out[10]: 2
```

```python
In [11]: df2.loc["a", "A"]  # a Series
Out[11]:
   A
a 0
a 1
Name: A, dtype: int64
```

### 2.11.2 Duplicate Label Detection

You can check whether an `Index` (storing the row or column labels) is unique with `Index.is_unique`:

```python
In [13]: df2
Out[13]:
   A
a 0
a 1
b 2
```

```python
In [14]: df2.index.is_unique
Out[14]: False
```

```python
In [15]: df2.columns.is_unique
Out[15]: True
```

**Note:** Checking whether an index is unique is somewhat expensive for large datasets. pandas does cache this result, so re-checking on the same index is very fast.

`Index.duplicated()` will return a boolean ndarray indicating whether a label is repeated.

```python
In [16]: df2.index.duplicated()
Out[16]: array([False, True, False])
```

Which can be used as a boolean filter to drop duplicate rows.
If you need additional logic to handle duplicate labels, rather than just dropping the repeats, using `groupby()` on the index is a common trick. For example, we’ll resolve duplicates by taking the average of all rows with the same label.

```python
In [17]: df2.loc[~df2.index.duplicated(), :]
Out[17]:
   A   
a  0  
b  2
```

2.11.3 Disallowing Duplicate Labels

New in version 1.2.0.

As noted above, handling duplicates is an important feature when reading in raw data. That said, you may want to avoid introducing duplicates as part of a data processing pipeline (from methods like `pandas.concat()`, `rename()`, etc.). Both `Series` and `DataFrame` disallow duplicate labels by calling `.set_flags(allow_duplicates_labels=False)`. (the default is to allow them). If there are duplicate labels, an exception will be raised.

```python
In [19]: pd.Series([0, 1, 2], index=["a", "b", "b"]).set_flags(allow_duplicates_labels=False)
---------------------------------------------------------------------------
DuplicateLabelError Traceback (most recent call last)
  Input In [19], in <cell line: 1>()
    return df

File /pandas/pandas/core/generic.py:438, in NDFrame.set_flags(self, copy, allows_duplicates_labels)
    436 df = self.copy(deep=copy)
    437 if allows_duplicates_labels is not None:
--> 438 df.flags["allows_duplicates_labels"] = allows_duplicates_labels
    439 return df

File /pandas/pandas/core/flags.py:105, in Flags.__setitem__(self, key, value)
    92 ax._maybe_check_unique()  # raises ValueError if not unique
--> 93 self._flags[k] = v

File /pandas/pandas/core/flags.py:105, in Flags.__setitem__(self, key, value)
    103 if key not in self._keys:
    104     raise ValueError(f"Unknown flag \{key}. Must be one of \{self._keys}\")
--> 105 setattr(self, key, value)

File /pandas/pandas/core/flags.py:92, in Flags.allows_duplicates_labels(self, value)
    90 if not value:
    91     for ax in obj.axes:
--> 92         ax._maybe_check_unique()
    93     self._allows_duplicates_labels = value

File /pandas/pandas/core/indexes/base.py:715, in Index._maybe_check_unique(self)
(continues on next page)
This applies to both row and column labels for a DataFrame

```
In [20]: df = pd.DataFrame(["A", "B", "C"], columns=[0, 1, 2, 3, 4, 5], index=["A", "B", "C"], allows_duplicate_labels=False)
```

```
Out[20]:
    A B C
0 0 1 2
1 3 4 5
```

This attribute can be checked or set with `allows_duplicate_labels`, which indicates whether that object can have duplicate labels.

```
In [21]: df = pd.DataFrame(["A", [0, 1, 2, 3], index=["A", "B", "C"], allows_duplicate_labels=False)
```

```
In [22]: df
```

```
Out[22]:
    A
x 0
y 1
```

```
In [23]: df.flags.allows_duplicate_labels
```

```
Out[23]: False
```

```
DataFrame.set_flags() can be used to return a new DataFrame with attributes like `allows_duplicate_labels` set to some value

```
In [24]: df2 = df.set_flags(allows_duplicate_labels=True)
```

```
In [25]: df2.flags.allows_duplicate_labels
Out[25]: True
```

The new DataFrame returned is a view on the same data as the old DataFrame. Or the property can just be set directly on the same object

```
In [26]: df2.flags.allows_duplicate_labels = False
```
When processing raw, messy data you might initially read in the messy data (which potentially has duplicate labels),
deduplicate, and then disallow duplicates going forward, to ensure that your data pipeline doesn’t introduce duplicates.

```python
>>> raw = pd.read_csv("...")
>>> deduplicated = raw.groupby(level=0).first()  # remove duplicates
>>> deduplicated.flags.allows_duplicate_labels = False  # disallow going forward
```

Setting `allows_duplicate_labels=True` on a `Series` or `DataFrame` with duplicate labels or performing an operation that introduces duplicate labels on a `Series` or `DataFrame` that disallows duplicates will raise an `DuplicateLabelError`.

```python
In [28]: df.rename(str.upper)
```

```
---------------------------------------------------------------------------
DuplicateLabelError                                Traceback (most recent call last)
Input In [28], in <cell line: 1>()
-----> 1 df.rename(str.upper)

File /pandas/pandas/core/frame.py:5083, in DataFrame.rename(self, mapper, index, columns,
→ axis, copy, inplace, level, errors)
    5083     return super()._rename(
    5084         mapper=mapper,
--> 5085         index=index,
    5086         columns=columns,
    5087         axis=axis,
    5088         copy=copy,
    5089         inplace=inplace,
    5090         level=level,
    5091         errors=errors,
    5092     )

File /pandas/pandas/core/generic.py:1163, in NDFrame._rename(self, mapper, index,
→ columns, axis, copy, inplace, level, errors)
    1163     return None
    1164     else:
--> 1163         return result.__finalize__(self, method="rename")

File /pandas/pandas/core/generic.py:5541, in NDFrame.__finalize__(self, other, method,
→ *, **kwargs)
```

(continues on next page)
This error message contains the labels that are duplicated, and the numeric positions of all the duplicates (including the “original”) in the Series or DataFrame.

**Duplicate Label Propagation**

In general, disallowing duplicates is “sticky”. It’s preserved through operations.

```python
In [29]: s1 = pd.Series(0, index=["a", "b"]).set_flags(allows_duplicate_labels=False)
In [30]: s1
Out[30]:
a    0
b    0
dtype: int64
```

```python
In [31]: s1.head().rename({"a": "b"})
```

```
DuplicateLabelError: Index has duplicates.
	positions

label

X    [0, 2]
Y    [1, 3]
```

```
File /pandas/pandas/core/indexes/base.py:715, in Index._maybe_check_unique(self)
    712 duplicates = self._format_duplicate_message()
    713 msg += f"\n{duplicates}"
--> 715 raise DuplicateLabelError(msg)

DuplicateLabelError: Index has duplicates.
	positions

label

X    [0, 2]
Y    [1, 3]
```

```
File /pandas/pandas/core/series.py:4601, in Series.rename(self, index, axis, copy, inplace, level, errors)
    4598     axis = self._get_axis_number(axis)
    4600     if callable(index) or is_dict_like(index):
--> 4601         return super().rename(

(continues on next page)
2.12 Categorical data

This is an introduction to pandas categorical data type, including a short comparison with R's factor.

Categoricals are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (categories; levels in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

In contrast to statistical categorical variables, categorical data might have an order (e.g. 'strongly agree' vs 'agree' or 'first observation' vs. 'second observation'), but numerical operations (additions, divisions, ...) are not possible.

All values of categorical data are either in categories or np.nan. Order is defined by the order of categories, not lexical order of the values. Internally, the data structure consists of a categories array and an integer array of codes which point to the real value in the categories array.
The categorical data type is useful in the following cases:

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory, see here.
- The lexical order of a variable is not the same as the logical order (“one”, “two”, “three”). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order, see here.
- As a signal to other Python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

See also the API docs on categoricals.

### 2.12.1 Object creation

#### Series creation

Categorical Series or columns in a DataFrame can be created in several ways:

By specifying `dtype="category"` when constructing a Series:

```python
In [1]: s = pd.Series(["a", "b", "c", "a"], dtype="category")
```

```python
In [2]: s
Out[2]:
0   a
1   b
2   c
3   a
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

By converting an existing Series or column to a category dtype:

```python
In [3]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})
```

```python
In [4]: df["B"] = df["A"].astype("category")
```

```python
In [5]: df
Out[5]:
     A     B
0   a   a
1   b   b
2   c   c
3   a   a
```

By using special functions, such as `cut()`, which groups data into discrete bins. See the example on tiling in the docs.

```python
In [6]: df = pd.DataFrame({"value": np.random.randint(0, 100, 20)})
```

```python
In [7]: labels = ["\{0\} - \{1\}".format(i, i + 9) for i in range(0, 100, 10)]
```

```python
In [8]: df["group"] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)
```
By passing a `pandas.Categorical` object to a Series or assigning it to a DataFrame.

```python
In [10]: raw_cat = pd.Categorical(
    ....:     ['a', 'b', 'c', 'a'], categories=['b', 'c', 'd'], ordered=False
    ....: )

In [11]: s = pd.Series(raw_cat)

In [12]: s
Out[12]:
0   NaN
1     b
2     c
3   NaN
dtype: category
Categories (3, object): ['b', 'c', 'd']

In [13]: df = pd.DataFrame({'A': ['a', 'b', 'c', 'a']})

In [14]: df['B'] = raw_cat

In [15]: df
Out[15]:
   A  B
0  a  NaN
1  b     b
2  c     c
3  a   NaN
```

Categorical data has a specific category dtype:

```python
In [16]: df.dtypes
Out[16]:
A  object
B  category
dtype: object
```
DataFrame creation

Similar to the previous section where a single column was converted to categorical, all columns in a DataFrame can be batch converted to categorical either during or after construction.

This can be done during construction by specifying dtype="category" in the DataFrame constructor:

```python
In [17]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')}, dtype='category')
```
```
In [18]: df.dtypes
Out[18]:
A    category
B    category
dtype: object
```

Note that the categories present in each column differ; the conversion is done column by column, so only labels present in a given column are categories:

```python
In [19]: df['A']
Out[19]:
0  a
1  b
2  c
3  a
Name: A, dtype: category
Categories (3, object): ['a', 'b', 'c']
```
```
In [20]: df['B']
Out[20]:
0  b
1  c
2  c
3  d
Name: B, dtype: category
Categories (3, object): ['b', 'c', 'd']
```

Analogously, all columns in an existing DataFrame can be batch converted using DataFrame.astype():

```python
In [21]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})
```
```
In [22]: df_cat = df.astype('category')
```
```
In [23]: df_cat.dtypes
Out[23]:
A    category
B    category
dtype: object
```

This conversion is likewise done column by column:

```python
In [24]: df_cat['A']
Out[24]:
0  a
1  b
(continues on next page)```
Controlling behavior

In the examples above where we passed `dtype='category'`, we used the default behavior:

1. Categories are inferred from the data.
2. Categories are unordered.

To control those behaviors, instead of passing 'category', use an instance of `CategoricalDtype`.

```python
In [26]: from pandas.api.types import CategoricalDtype
In [27]: cat_type = CategoricalDtype(categories=list("abcd"), ordered=True)
In [28]: df_cat = df.astype(cat_type)
In [29]: df_cat["A"]
```

Similarly, a `CategoricalDtype` can be used with a DataFrame to ensure that categories are consistent among all columns.

```python
In [31]: from pandas.api.types import CategoricalDtype
In [32]: df = pd.DataFrame({"A": list("abca"), "B": list("bcd"))
In [33]: cat_type = CategoricalDtype(categories=list("abcd"), ordered=True)
In [34]: df_cat = df.astype(cat_type)
In [35]: df_cat["A"]
```
Out[35]:
0   a
1   b
2   c
3   a
Name: A, dtype: category
Categories (4, object): ['a' < 'b' < 'c' < 'd']

In [36]: df_cat['B']
Out[36]:
0   b
1   c
2   c
3   d
Name: B, dtype: category
Categories (4, object): ['a' < 'b' < 'c' < 'd']

**Note:** To perform table-wise conversion, where all labels in the entire DataFrame are used as categories for each column, the categories parameter can be determined programmatically by categories = pd.unique(df.to_numpy().ravel()).

If you already have codes and categories, you can use the `from_codes()` constructor to save the factorize step during normal constructor mode:

In [37]: splitter = np.random.choice([0, 1], 5, p=[0.5, 0.5])
In [38]: s = pd.Series(pd.Categorical.from_codes(splitter, categories=['train', 'test']))

**Regaining original data**

To get back to the original Series or NumPy array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:

In [39]: s = pd.Series(['a', 'b', 'c', 'a'])
In [40]: s
Out[40]:
0   a
1   b
2   c
3   a
dtype: object

In [41]: s2 = s.astype("category")
In [42]: s2
Out[42]:
0   a
1   b
2   c
3  a
dtype: category
Categories (3, object): ['a', 'b', 'c']

In [43]: s2.astype(str)
Out[43]:
0  a
1  b
2  c
3  a
dtype: object

In [44]: np.asarray(s2)
Out[44]:
array(['a', 'b', 'c', 'a'], dtype=object)

Note: In contrast to R's `factor` function, categorical data is not converting input values to strings; categories will end up the same data type as the original values.

Note: In contrast to R's `factor` function, there is currently no way to assign/change labels at creation time. Use `categories` to change the categories after creation time.

### 2.12.2 CategoricalDtype

A categorical's type is fully described by

1. **categories**: a sequence of unique values and no missing values
2. **ordered**: a boolean

This information can be stored in a `CategoricalDtype`. The `categories` argument is optional, which implies that the actual categories should be inferred from whatever is present in the data when the `pandas.Categorical` is created. The categories are assumed to be unordered by default.

In [45]: from pandas.api.types import CategoricalDtype

In [46]: CategoricalDtype(['a', 'b', 'c'], ordered=False)
Out[46]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)

In [47]: CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[47]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [48]: CategoricalDtype()
Out[48]: CategoricalDtype(categories=None, ordered=False)

A `CategoricalDtype` can be used in any place pandas expects a `dtype`. For example `pandas.read_csv()`, `pandas.DataFrame.astype()`, or in the `Series` constructor.

Note: As a convenience, you can use the string 'category' in place of a `CategoricalDtype` when you want the default behavior of the categories being unordered, and equal to the set values present in the array. In other words,
Equality semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and order. When comparing two unordered categoricals, the order of the categories is not considered.

```
In [49]: c1 = CategoricalDtype(["a", "b", "c"], ordered=False)

# Equal, since order is not considered when ordered=False
In [50]: c1 == CategoricalDtype(["b", "c", "a"], ordered=False)
Out[50]: True

# Unequal, since the second CategoricalDtype is ordered
In [51]: c1 == CategoricalDtype(["a", "b", "c"], ordered=True)
Out[51]: False
```

All instances of `CategoricalDtype` compare equal to the string 'category'.

```
In [52]: c1 == "category"
Out[52]: True
```

**Warning:** Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances of `CategoricalDtype` compare equal to 'category', all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of categories or ordered.

2.12.3 Description

Using `describe()` on categorical data will produce similar output to a `Series` or `DataFrame` of type string.

```
In [53]: cat = pd.Categorical(['a', 'c', 'c', np.nan], categories=['b', 'a', 'c'])
In [54]: df = pd.DataFrame({'cat': cat, 's': ['a', 'c', 'c', np.nan]})
In [55]: df.describe()
Out[55]:
   cat    s
  count   3   3
  unique  2   2
  top     c   c
  freq    2   2

In [56]: df['cat'].describe()
Out[56]:
  count   3
  unique  2
  top     c
  freq    2
Name: cat, dtype: object
```
### 2.12.4 Working with categories

Categorical data has a `categories` and an `ordered` property, which list their possible values and whether the ordering matters or not. These properties are exposed as `s.cat.categories` and `s.cat.ordered`. If you don't manually specify categories and ordering, they are inferred from the passed arguments.

```python
In [57]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [58]: s.cat.categories
Out[58]: Index(['a', 'b', 'c'], dtype='object')

In [59]: s.cat.ordered
Out[59]: False
```

It's also possible to pass in the categories in a specific order:

```python
In [60]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], categories=["c", "b", "a"]))

In [61]: s.cat.categories
Out[61]: Index(['c', 'b', 'a'], dtype='object')

In [62]: s.cat.ordered
Out[62]: False
```

**Note:** New categorical data are not automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered Categorical.

**Note:** The result of `unique()` is not always the same as `Series.cat.categories`, because `Series.unique()` has a couple of guarantees, namely that it returns categories in the order of appearance, and it only includes values that are actually present.

```python
In [63]: s = pd.Series(list("babc")).astype(CategoricalDtype(list("abcd")))

In [64]: s
Out[64]:
0   b
1   a
2   b
3   c
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

# categories
In [65]: s.cat.categories
Out[65]: Index(['a', 'b', 'c', 'd'], dtype='object')

# uniques
In [66]: s.unique()
Out[66]:
['b', 'a', 'c']
Categories (4, object): ['a', 'b', 'c', 'd']
```
Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `rename_categories()` method:

```python
In [67]: s = pd.Series(['a', 'b', 'c', 'a'], dtype='category')

In [68]: s
Out[68]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): ['a', 'b', 'c']

In [69]: s.cat.categories = ['Group %s' % g for g in s.cat.categories]

In [70]: s
Out[70]:
0  Group a
1  Group b
2  Group c
3  Group a
dtype: category
Categories (3, object): ['Group a', 'Group b', 'Group c']

In [71]: s = s.cat.rename_categories([1, 2, 3])

In [72]: s
Out[72]:
0     1
1     2
2     3
3     1
dtype: category
Categories (3, int64): [1, 2, 3]

# You can also pass a dict-like object to map the renaming
In [73]: s = s.cat.rename_categories({1: "x", 2: "y", 3: "z"})

In [74]: s
Out[74]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): ['x', 'y', 'z']
```

**Note:** In contrast to R’s `factor`, categorical data can have categories of other types than string.
**Note:** Be aware that assigning new categories is an in-place operation, while most other operations under `Series.cat` per default return a new `Series` of dtype `category`.

Categories must be unique or a `ValueError` is raised:

```
In [75]: try:
    ....:     s.cat.categories = [1, 1, 1]
    ....: except ValueError as e:
    ....:     print("ValueError:", str(e))
    ....:
ValueError: Categorical categories must be unique
```

Categories must also not be `NaN` or a `ValueError` is raised:

```
In [76]: try:
    ....:     s.cat.categories = [1, 2, np.nan]
    ....: except ValueError as e:
    ....:     print("ValueError:", str(e))
    ....:
ValueError: Categorical categories cannot be null
```

**Appending new categories**

Appending categories can be done by using the `add_categories()` method:

```
In [77]: s = s.cat.add_categories([4])

In [78]: s.cat.categories
Out[78]: Index(['x', 'y', 'z', 4], dtype='object')

In [79]: s
Out[79]:
0  x
1  y
2  z
3  x
dtype: category
Categories (4, object): ['x', 'y', 'z', 4]
```

**Removing categories**

Removing categories can be done by using the `remove_categories()` method. Values which are removed are replaced by `np.nan`:

```
In [80]: s = s.cat.remove_categories([4])

In [81]: s
Out[81]:
0  x
1  y
```

(continues on next page)
Removing unused categories

Removing unused categories can also be done:

```python
In [82]: s = pd.Series(pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c', 'd']))

In [83]: s
Out[83]:
0  a
1  b
2  a
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

In [84]: s.cat.remove_unused_categories()
Out[84]:
0  a
1  b
2  a
dtype: category
Categories (2, object): ['a', 'b']
```

Setting categories

If you want to do remove and add new categories in one step (which has some speed advantage), or simply set the categories to a predefined scale, use `set_categories()`.

```python
In [85]: s = pd.Series(['one', 'two', 'four', '-'], dtype='category')

In [86]: s
Out[86]:
0  one
1  two
2  four
3  -
dtype: category
Categories (4, object): ['-', 'four', 'one', 'two']

In [87]: s = s.cat.set_categories(['one', 'two', 'three', 'four'])

In [88]: s
Out[88]:
0  one
1  two
2  four
```
Note: Be aware that `Categorical.set_categories()` cannot know whether some category is omitted intentionally or because it is misspelled or (under Python3) due to a type difference (e.g., NumPy S1 dtype and Python strings). This can result in surprising behaviour!

### 2.12.5 Sorting and order

If categorical data is ordered (`s.cat.ordered == True`), then the order of the categories has a meaning and certain operations are possible. If the categorical is unordered, `.min()`/.`max()` will raise a `TypeError`.

```python
In [89]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], ordered=False))
In [90]: s.sort_values(inplace=True)
In [91]: s = pd.Series(["a", "b", "c", "a"]).astype(CategoricalDtype(ordered=True))
In [92]: s.sort_values(inplace=True)
In [93]: s
Out[93]:
0  a
3  a
1  b
2  c
dtype: category
Categories (3, object): ['a' < 'b' < 'c']
```

You can set categorical data to be ordered by using `as_ordered()` or unordered by using `as_unordered()`. These will by default return a `new` object.

```python
In [95]: s.cat.as_ordered()
Out[95]:
0  a
3  a
1  b
2  c
dtype: category
Categories (3, object): ['a' < 'b' < 'c']
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

1    b
dtype: category
Categories (3, object): ['a', 'b', 'c']

Sorting will use the order defined by categories, not any lexical order present on the data type. This is even true for strings and numeric data:

```
In [97]: s = pd.Series([1, 2, 3, 1], dtype="category")
In [98]: s = s.cat.set_categories([2, 3, 1], ordered=True)
In [99]: s
Out[99]:
0  1
1  2
2  3
3  1
dtype: category
Categories (3, int64): [2 < 3 < 1]
In [100]: s.sort_values(inplace=True)
In [101]: s
Out[101]:
1  2
2  3
0  1
3  1
dtype: category
Categories (3, int64): [2 < 3 < 1]
In [102]: s.min(), s.max()
Out[102]: (2, 1)
```

Reordering

Reordering the categories is possible via the Categorical.reorder_categories() and the Categorical.set_categories() methods. For Categorical.reorder_categories(), all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [103]: s = pd.Series([1, 2, 3, 1], dtype="category")
In [104]: s = s.cat.reorder_categories([2, 3, 1], ordered=True)
In [105]: s
Out[105]:
0  1
1  2
2  3
```
In [106]: s.sort_values(inplace=True)

In [107]: s
Out[107]:
   0  1
   1  2
   2  3
   3  1

Note: Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the Series, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the Series are changed.

Note: If the Categorical is not ordered, Series.min() and Series.max() will raise TypeError. Numeric operations like +, -, *, / and operations based on them (e.g. Series.median(), which would need to compute the mean between two values if the length of an array is even) do not work and raise a TypeError.

Multi column sorting

A categorical dtyped column will participate in a multi-column sort in a similar manner to other columns. The ordering of the categorical is determined by the categories of that column.

```python
In [109]: dfs = pd.DataFrame(
    ...:     {  
    ...:         "A": pd.Categorical(  
    ...:             list("bbeebbaa"),  
    ...:             categories=["e", "a", "b"],  
    ...:             ordered=True,  
    ...:         ),  
    ...:         "B": [1, 2, 1, 2, 2, 1, 2, 1],  
    ...:     }  
    ...: )

In [110]: dfs.sort_values(by=["A", "B"])
Out[110]:
    A  B
2  e  1
```

(continues on next page)
Reordering the categories changes a future sort.

```
In [111]: dfs['A'] = dfs['A'].cat.reorder_categories(['a', 'b', 'e'])
```

```
In [112]: dfs.sort_values(by=['A', 'B'])
Out[112]:
   A  B
0 b 1
1 b 2
2 e 1
3 e 2
4 b 2
5 b 1
6 a 2
7 a 1
```

### 2.12.6 Comparisons

Comparing categorical data with other objects is possible in three cases:

- Comparing equality (== and !=) to a list-like object (list, Series, array, ...) of the same length as the categorical data.
- All comparisons (==, !=, >, >=, <, and <=) of categorical data to another categorical Series, when ordered==True and the categories are the same.
- All comparisons of a categorical data to a scalar.

All other comparisons, especially “non-equality” comparisons of two categoricals with different categories or a categorical with any list-like object, will raise a `TypeError`.

**Note:** Any “non-equality” comparisons of categorical data with a `Series`, `np.array`, `list` or categorical data with different categories or ordering will raise a `TypeError` because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [113]: cat = pd.Series([1, 2, 3]).astype(CategoricalDtype([3, 2, 1], ordered=True))
```

```
In [114]: cat_base = pd.Series([2, 2, 2]).astype(CategoricalDtype([3, 2, 1],_,ordered=True))
```

```
In [115]: cat_base2 = pd.Series([2, 2, 2]).astype(CategoricalDtype(ordered=True))
```

```
In [116]: cat
```
Comparing to a categorical with the same categories and ordering or to a scalar works:

In [119]: cat > cat_base
Out[119]:
0   True
1   False
2   False
dtype: bool

In [120]: cat > 2
Out[120]:
0   True
1   False
2   False
dtype: bool

Equality comparisons work with any list-like object of same length and scalars:

In [121]: cat == cat_base
Out[121]:
0   False
1    True
2   False
dtype: bool

In [122]: cat == np.array([1, 2, 3])
Out[122]:
0    True
1    True
This doesn’t work because the categories are not the same:

```python
In [124]: try:
    ....:     cat > cat_base2
    ....: except TypeError as e:
    ....:     print("TypeError: ", str(e))
    ....:
TypeError: Categoricals can only be compared if 'categories' are the same.
```

If you want to do a “non-equality” comparison of a categorical series with a list-like object which is not categorical data, you need to be explicit and convert the categorical data back to the original values:

```python
In [125]: base = np.array([1, 2, 3])

In [126]: try:
    ....:     cat > base
    ....: except TypeError as e:
    ....:     print("TypeError: ", str(e))
    ....:
TypeError: Cannot compare a Categorical for op __gt__ with type <class 'numpy.ndarray'>. If you want to compare values, use 'np.asarray(cat) <op> other'.
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```python
In [128]: c1 = pd.Categorical(["a", "b"], categories=["a", "b"], ordered=False)

In [129]: c2 = pd.Categorical(["a", "b"], categories=["b", "a"], ordered=False)

In [130]: c1 == c2
Out[130]: array([ True,  True])
```
2.12.7 Operations

Apart from `Series.min()`, `Series.max()` and `Series.mode()`, the following operations are possible with categorical data:

Series methods like `Series.value_counts()` will use all categories, even if some categories are not present in the data:

```python
In [131]: s = pd.Series(pd.Categorical(
                      ["a", "b", "c", "c"], categories=["c", "a", "b", 
                      "d"]))
In [132]: s.value_counts()
Out[132]:
c 2
a 1
b 1
d 0
dtype: int64
```

DataFrame methods like `DataFrame.sum()` also show “unused” categories.

```python
In [133]: columns = pd.Categorical(
                      .....: ["One", "One", "Two"], categories=["One", "Two", "Three"], ordered=True
                      .....: )
                      .....:
In [134]: df = pd.DataFrame(
                      .....: data=[[1, 2, 3], [4, 5, 6]],
                      .....: columns=pd.MultiIndex.from_arrays([["A", "B", "B"], columns]),
                      .....: )
                      .....:
In [135]: df.groupby(axis=1, level=1).sum()
Out[135]:
        One  Two  Three
0      3   3     0
1      9   6     0
```

Groupby will also show “unused” categories:

```python
In [136]: cats = pd.Categorical(
                      .....: ["a", "b", "b", "b", "c", "c", "c"], categories=["a", "b", "c", "d"]
                      .....: )
                      .....:
In [137]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 3, 4, 5]})
In [138]: df.groupby("cats").mean()
Out[138]:
        values
cats
a     1.0
b     2.0
c     4.0
```

(continues on next page)
In [139]: cats2 = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [140]: df2 = pd.DataFrame(
.....:     {  
.....:         "cats": cats2,  
.....:         "B": ["c", "d", "c", "d"],  
.....:         "values": [1, 2, 3, 4],  
.....:     }  
.....: )

In [141]: df2.groupby(["cats", "B"]).mean()

Out[141]:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>2.0</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>4.0</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Pivot tables:

In [142]: raw_cat = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [143]: df = pd.DataFrame({"A": raw_cat, "B": ["c", "d", "c", "d"], "values": [1, 2, 3, 4]})

In [144]: pd.pivot_table(df, values="values", index=["A", "B"])

Out[144]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
2.12.8 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.at`, and `.iat` work as normal. The only difference is the return type (for getting) and that only values already in `categories` can be assigned.

**Getting**

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.

```python
In [145]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])
In [146]: cats = pd.Series(["a", "b", "b", "b", "c", "c", "c"], dtype="category", index=idx)
In [147]: values = [1, 2, 2, 2, 3, 4, 5]
In [148]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)
In [149]: df.iloc[2:4, :]
Out[149]:
   cats  values
j  b    2
k  b    2
In [150]: df.iloc[2:4, :].dtypes
Out[150]:
cats    category
values  int64
dtype: object
In [151]: df.loc["h":"j", "cats"]
Out[151]:
   cats
h  a
i  b
j  b
Name: cats, dtype: category
Categories (3, object): ['a', 'b', 'c']
In [152]: df[df["cats"] == "b"]
Out[152]:
   cats  values
  i  b    2
  j  b    2
  k  b    2
```

An example where the category type is not preserved is if you take one single row: the resulting `Series` is of dtype `object`:

```python
# get the complete "h" row as a Series
In [153]: df.loc["h", :]
Out[153]:
   cats
h  a
(continues on next page)
```

2.12. Categorical data
Returning a single item from categorical data will also return the value, not a categorical of length “1”.

```python
In [154]: df.iat[0, 0]
Out[154]: 'a'

In [155]: df["cats"].cat.categories = ["x", "y", "z"]

In [156]: df.at["h", "cats"]  # returns a string
Out[156]: 'x'
```

**Note:** The is in contrast to R’s `factor` function, where `factor(c(1,2,3))[1]` returns a single value `factor`.

To get a single value Series of type `category`, you pass in a list with a single value:

```python
In [157]: df.loc["h", "cats"]
Out[157]:
   h
Name: cats, dtype: category
Categories (3, object): ['x', 'y', 'z']
```

**String and datetime accessors**

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```python
In [158]: str_s = pd.Series(list("aabb"))

In [159]: str_cat = str_s.astype("category")

In [160]: str_cat
Out[160]:
   0 a
   1 a
   2 b
   3 b
dtype: category
Categories (2, object): ['a', 'b']

In [161]: str_cat.str.contains("a")
Out[161]:
   0   True
   1   True
   2  False
   3  False
dtype: bool

In [162]: date_s = pd.Series(pd.date_range("1/1/2015", periods=5))
```

(continues on next page)
In [163]: date_cat = date_s.astype("category")

In [164]: date_cat

Out[164]:
0 2015-01-01
1 2015-01-02
2 2015-01-03
3 2015-01-04
4 2015-01-05
dtype: category

In [165]: date_cat.dt.day

Out[165]:
0 1
1 2
2 3
3 4
4 5
dtype: int64

Note: The returned Series (or DataFrame) is of the same type as if you used the .str.<method>/ .dt.<property> on a Series of that type (and not of type category!).

That means, that the returned values from methods and properties on the accessors of a Series and the returned values from methods and properties on the accessors of this Series transformed to one of type category will be equal:

In [166]: ret_s = str_s.str.contains("a")

In [167]: ret_cat = str_cat.str.contains("a")

In [168]: ret_s.dtype == ret_cat.dtype
Out[168]: True

In [169]: ret_s == ret_cat
Out[169]:
0 True
1 True
2 True
3 True
dtype: bool

Note: The work is done on the categories and then a new Series is constructed. This has some performance implication if you have a Series of type string, where lots of elements are repeated (i.e. the number of unique elements in the Series is a lot smaller than the length of the Series). In this case it can be faster to convert the original Series to one of type category and use .str.<method> or .dt.<property> on that.
Setting

Setting values in a categorical column (or Series) works as long as the value is included in the categories:

```python
In [170]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [171]: cats = pd.Categorical(["a", "a", "a", "a", "a", "a", "a"], categories=["a", "b")

In [172]: values = [1, 1, 1, 1, 1, 1]

In [173]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [174]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

  Out[175]:
  cats  values
  h    a   1
  i    a   1
  j    b   2
  k    b   2
  l    a   1
  m    a   1
  n    a   1

In [176]: try:
   ....:    df.iloc[2:4, :] = [["c", 3], ["c", 3]]
   ....: except TypeError as e:
   ....:     print("TypeError:", str(e))
   ....:  TypeError: Cannot setitem on a Categorical with a new category, set the categories first
```

Setting values by assigning categorical data will also check that the categories match:

```python
In [177]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"], categories=["a", "b")

In [178]: df
  Out[178]:
  cats  values
  h    a   1
  i    a   1
  j    a   2
  k    a   2
  l    a   1
  m    a   1
  n    a   1

In [179]: try:
   ....:    df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"], categories=["a", "b", "c")
   ....: except TypeError as e:
   ....:     print("TypeError:", str(e))
   ....:
```

(continues on next page)
Assigning a Categorical to parts of a column of other types will use the values:

```python
In [180]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a", "a"]})
In [181]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])
In [182]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])
In [183]: df
Out[183]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a
```

```python
In [184]: df.dtypes
Out[184]:
   a object
  b object
dtype: object
```

### Merging / concatenation

By default, combining Series or DataFrames which contain the same categories results in category dtype, otherwise results will depend on the dtype of the underlying categories. Merges that result in non-categorical dtypes will likely have higher memory usage. Use .astype or union_categoricals to ensure category results.

```python
In [185]: from pandas.api.types import union_categoricals

# same categories
In [186]: s1 = pd.Series(["a", "b"], dtype="category")
In [187]: s2 = pd.Series(["a", "b", "a"], dtype="category")
In [188]: pd.concat([s1, s2])
Out[188]:
  0  a
  1  b
  0  a
  1  b
  2  a
dtype: category
Categories (2, object): ['a', 'b']
```

# different categories
```python
In [189]: s3 = pd.Series(["b", "c"], dtype="category")
```
In [190]: pd.concat([s1, s3])  
Out[190]:
     0  a
     1  b
     0  b
     1  c
 dtype: object

# Output dtype is inferred based on categories values
In [191]: int_cats = pd.Series([1, 2], dtype="category")

In [192]: float_cats = pd.Series([3.0, 4.0], dtype="category")

In [193]: pd.concat([int_cats, float_cats])
Out[193]:
       0
0  1.0
1  2.0
0  3.0
1  4.0
 dtype: float64

In [194]: pd.concat([s1, s3]).astype("category")
Out[194]:
     0  a
     1  b
     0  b
     1  c
dtype: category
Categories (3, object): ['a', 'b', 'c']

In [195]: union_categoricals([s1.array, s3.array])
Out[195]: ['a', 'b', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']

The following table summarizes the results of merging Categoricals:

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>identical</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>category</td>
<td>True</td>
<td>category</td>
</tr>
<tr>
<td>category (object)</td>
<td>category (object)</td>
<td>False</td>
<td>object (dtype is inferred)</td>
</tr>
<tr>
<td>category (int)</td>
<td>category (float)</td>
<td>False</td>
<td>float (dtype is inferred)</td>
</tr>
</tbody>
</table>

See also the section on *merge dtypes* for notes about preserving merge dtypes and performance.
Unioning

If you want to combine categoricals that do not necessarily have the same categories, the `union_categoricals()` function will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```python
In [196]: from pandas.api.types import union_categoricals

In [197]: a = pd.Categorical(['b', 'c'])

In [198]: b = pd.Categorical(['a', 'b'])

In [199]: union_categoricals([a, b])
Out[199]:
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']
```

By default, the resulting categories will be ordered as they appear in the data. If you want the categories to be lexsorted, use `sort_categories=True` argument.

```python
In [200]: union_categoricals([a, b], sort_categories=True)
Out[200]:
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']
```

`union_categoricals` also works with the “easy” case of combining two categoricals of the same categories and order information (e.g. what you could also `append` for).

```python
In [201]: a = pd.Categorical(['a', 'b'], ordered=True)

In [202]: b = pd.Categorical(['a', 'b', 'a'], ordered=True)

In [203]: union_categoricals([a, b])
Out[203]:
['a', 'b', 'a', 'b']
Categories (2, object): ['a' < 'b']
```

The below raises `TypeError` because the categories are ordered and not identical.

```python
In [1]: a = pd.Categorical(['a', 'b'], ordered=True)
In [2]: b = pd.Categorical(['a', 'b', 'c'], ordered=True)
In [3]: union_categoricals([a, b])
Out[3]:
TypeError: to union ordered Categoricals, all categories must be the same
```

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```python
In [204]: a = pd.Categorical(['a', 'b', 'c'], ordered=True)

In [205]: b = pd.Categorical(['c', 'b', 'a'], ordered=True)

In [206]: union_categoricals([a, b], ignore_order=True)
Out[206]:
['a', 'b', 'c', 'c', 'b', 'a']
Categories (3, object): ['a', 'b', 'c']
```
union_categoricals() also works with a CategoricalIndex, or Series containing categorical data, but note that the resulting array will always be a plain Categorical:

```python
In [207]: a = pd.Series(["b", "c"], dtype="category")
In [208]: b = pd.Series(["a", "b"], dtype="category")
In [209]: union_categoricals([a, b])
Out[209]:
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']
```

Note: union_categoricals may recode the integer codes for categories when combining categoricals. This is likely what you want, but if you are relying on the exact numbering of the categories, be aware.

```python
In [210]: c1 = pd.Categorical(["b", "c"])
In [211]: c2 = pd.Categorical(["a", "b"])
In [212]: c1
Out[212]:
['b', 'c']
Categories (2, object): ['b', 'c']
# "b" is coded to 0
In [213]: c1.codes
Out[213]: array([0, 1], dtype=int8)
In [214]: c2
Out[214]:
['a', 'b']
Categories (2, object): ['a', 'b']
# "b" is coded to 1
In [215]: c2.codes
Out[215]: array([0, 1], dtype=int8)
In [216]: c = union_categoricals([c1, c2])
In [217]: c
Out[217]:
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']
# "b" is coded to 0 throughout, same as c1, different from c2
In [218]: c.codes
Out[218]: array([0, 1, 2, 0], dtype=int8)
```
2.12.9 Getting data in/out

You can write data that contains category dtypes to a HDFStore. See here for an example and caveats.

It is also possible to write data to and reading data from Stata format files. See here for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to category and assign the right categories and categories ordering.

```python
In [219]: import io

In [220]: s = pd.Series(pd.Categorical(
        ["a", "b", "b", "a", "a", "d"]))

# rename the categories
In [221]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [222]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [223]: df = pd.DataFrame({
    "cats": s,
    "vals": [1, 2, 3, 4, 5, 6]
})

In [224]: csv = io.StringIO()

In [225]: df.to_csv(csv)

In [226]: df2 = pd.read_csv(io.StringIO(csv.getvalue()))

In [227]: df2.dtypes
Out[227]:
        Unnamed: 0       int64
       cats            object
       vals          int64
dtype: object

In [228]: df2["cats"]
Out[228]:
0  very good
1    good
2    good
3  very good
4  very good
5     bad
Name: cats, dtype: object

# Redo the category
In [229]: df2["cats"] = df2["cats"].astype("category")

In [230]: df2["cats"].cat.set_categories(  
       ["very bad", "bad", "medium", "good", "very good"], inplace=True
   ......: )
   ......:

In [231]: df2.dtypes

(continues on next page)
Out[231]:
Unnamed: 0    int64
cats          category
vals          int64
dtype: object

In [232]: df2["cats"]
Out[232]:
0    very good
1     good
2     good
3   very good
4   very good
5      bad
Name: cats, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']

The same holds for writing to a SQL database with to_sql.

2.12.10 Missing data

pandas primarily uses the value np.nan to represent missing data. It is by default not included in computations. See the Missing Data section.

Missing values should not be included in the Categorical’s categories, only in the values. Instead, it is understood that NaN is different, and is always a possibility. When working with the Categorical’s codes, missing values will always have a code of -1.

In [233]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")

# only two categories
In [234]: s
Out[234]:
0    a
1    b
2   NaN
3    a
dtype: category
Categories (2, object): ['a', 'b']

In [235]: s.cat.codes
Out[235]:
0   0
1   1
2  -1
3   0
dtype: int8

Methods for working with missing data, e.g. isna(), fillna(), dropna(), all work normally:

In [236]: s = pd.Series(["a", "b", np.nan], dtype="category")

(continues on next page)
In [237]: s
Out[237]:
0   a
1   b
2  NaN
dtype: category
Categories (2, object): ['a', 'b']

In [238]: pd.isna(s)
Out[238]:
0   False
1   False
2    True
dtype: bool

In [239]: s.fillna("a")
Out[239]:
0   a
1   b
2   a
dtype: category
Categories (2, object): ['a', 'b']

2.12.11 Differences to R’s `factor`

The following differences to R’s `factor` functions can be observed:

- R’s `levels` are named `categories`.
- R’s `levels` are always of type string, while `categories` in pandas can be of any dtype.
- It’s not possible to specify labels at creation time. Use `s.cat.rename_categories(new_labels)` afterwards.
- In contrast to R’s `factor` function, using categorical data as the sole input to create a new categorical series will *not* remove unused categories but create a new categorical series which is equal to the passed in one!
- R allows for missing values to be included in its `levels` (pandas’ `categories`). pandas does not allow NaN categories, but missing values can still be in the `values`.

2.12.12 Gotchas

Memory usage

The memory usage of a Categorical is proportional to the number of categories plus the length of the data. In contrast, an object dtype is a constant times the length of the data.

In [240]: s = pd.Series(["foo", "bar"] * 1000)

# object dtype
In [241]: s.nbytes
Out[241]: 16000

(continues on next page)
# category dtype
In [242]: s.astype("category").nbytes
Out[242]: 2016

Note: If the number of categories approaches the length of the data, the Categorical will use nearly the same or more memory than an equivalent object dtype representation.

In [243]: s = pd.Series(["foo%04d" % i for i in range(2000)])

# object dtype
In [244]: s.nbytes
Out[244]: 16000

# category dtype
In [245]: s.astype("category").nbytes
Out[245]: 20000

Categorical is not a numpy array

Currently, categorical data and the underlying Categorical is implemented as a Python object and not as a low-level NumPy array dtype. This leads to some problems.

NumPy itself doesn’t know about the new dtype:

In [246]: try:
   ....:   np.dtype("category")
   ....:   except TypeError as e:
   ....:     print("TypeError:", str(e))
   ....:
TypeError: data type 'category' not understood

In [247]: dtype = pd.Categorical(["a"]).dtype

In [248]: try:
   ....:   np.dtype(dtype)
   ....:   except TypeError as e:
   ....:     print("TypeError:", str(e))
   ....:
TypeError: Cannot interpret 'CategoricalDtype(categories=['a'], ordered=False)' as a...

Dtype comparisons work:

In [249]: dtype == np.str_
Out[249]: False

In [250]: np.str_ == dtype
Out[250]: False

To check if a Series contains Categorical data, use hasattr(s, 'cat'): 

668 Chapter 2. User Guide
pandas: powerful Python data analysis toolkit, Release 1.4.2

Using NumPy functions on a Series of type category should not work as Categoricals are not numeric data (even in the case that .categories is numeric).

```python
In [253]: s = pd.Series(pd.Categorical([1, 2, 3, 4]))
In [254]: try:
   ....:     np.sum(s)
   ....: except TypeError as e:
   ....:     print("TypeError: ", str(e))
TypeError: 'Categorical' with dtype category does not support reduction 'sum'
```

**Note:** If such a function works, please file a bug at https://github.com/pandas-dev/pandas!

**dtype in apply**

pandas currently does not preserve the dtype in apply functions: If you apply along rows you get a Series of object dtype (same as getting a row -> getting one element will return a basic type) and applying along columns will also convert to object. NaN values are unaffected. You can use fillna to handle missing values before applying a function.

```python
In [255]: df = pd.DataFrame(
   ....:     {
   ....:         "a": [1, 2, 3, 4],
   ....:         "b": ["a", "b", "c", "d"],
   ....:         "cats": pd.Categorical([1, 2, 3, 2]),
   ....:     }
   ....: )
In [256]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[256]:
0 <class 'int'>
1 <class 'int'>
2 <class 'int'>
3 <class 'int'>
dtype: object
```

```python
In [257]: df.apply(lambda col: col.dtype, axis=0)
Out[257]:
a    int64
b     object
cats  category
dtype: object
```

2.12. Categorical data

669
Categorical index

A CategoricalIndex is a type of index that is useful for supporting indexing with duplicates. This is a container around a Categorical and allows efficient indexing and storage of an index with a large number of duplicated elements. See the advanced indexing docs for a more detailed explanation.

Setting the index will create a CategoricalIndex:

```
In [258]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])
In [259]: strings = ["a", "b", "c", "d"]
In [260]: values = [4, 2, 3, 1]
In [261]: df = pd.DataFrame({"strings": strings, "values": values}, index=cats)
In [262]: df.index
Out[262]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False, dtype='category')
```

# This now sorts by the categories order
```
In [263]: df.sort_index()
Out[263]:
   strings values
     4     d     1
     2     b     2
     3     c     3
     1     a     4
```

Side effects

Constructing a Series from a Categorical will not copy the input Categorical. This means that changes to the Series will in most cases change the original Categorical:

```
In [264]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])
In [265]: s = pd.Series(cat, name="cat")
In [266]: cat
Out[266]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
In [267]: s.iloc[0:2] = 10
In [268]: cat
Out[268]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
In [269]: df = pd.DataFrame(s)
In [270]: df["cat"].cat.categories = [1, 2, 3, 4, 5]
```
In 

Out:

[10, 10, 3, 10]

Categories (5, int64): [1, 2, 3, 4, 10]

Use copy=True to prevent such a behaviour or simply don’t reuse Categoricals:

In 

Out:

[1, 2, 3, 10]

Categories (5, int64): [1, 2, 3, 4, 10]

In 

Out:

[1, 2, 3, 10]

Categories (5, int64): [1, 2, 3, 4, 10]

Note: This also happens in some cases when you supply a NumPy array instead of a Categorical: using an int array (e.g. np.array([1, 2, 3, 4])) will exhibit the same behavior, while using a string array (e.g. np.array([“a”, “b”, “c”, “a”])) will not.

2.13 Nullable integer data type

Note: IntegerArray is currently experimental. Its API or implementation may change without warning.

Changed in version 1.0.0: Now uses pandas.NA as the missing value rather than numpy.nan.

In Working with missing data, we saw that pandas primarily uses NaN to represent missing data. Because NaN is a float, this forces an array of integers with any missing values to become floating point. In some cases, this may not matter much. But if your integer column is, say, an identifier, casting to float can be problematic. Some integers cannot even be represented as floating point numbers.
2.13.1 Construction

pandas can represent integer data with possibly missing values using `arrays<IntegerArray>`. This is an *extension types* implemented within pandas.

```python
In [1]: arr = pd.array([1, 2, None], dtype=pd.Int64Dtype())
In [2]: arr
Out[2]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

Or the string alias "Int64" (note the capital "I", to differentiate from NumPy's 'int64' dtype):

```python
In [3]: pd.array([1, 2, np.nan], dtype="Int64")
Out[3]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

All NA-like values are replaced with pandas.NA.

```python
In [4]: pd.array([1, 2, np.nan, None, pd.NA], dtype="Int64")
Out[4]:
<IntegerArray>
[1, 2, <NA>, <NA>, <NA>]
Length: 5, dtype: Int64
```

This array can be stored in a `DataFrame` or `Series` like any NumPy array.

```python
In [5]: pd.Series(arr)
Out[5]:
0 1
1 2
2 <NA>
dtype: Int64
```

You can also pass the list-like object to the `Series` constructor with the dtype.

```python
In [6]: pd.array([1, None])
Out[6]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64
```

```python
In [7]: pd.array([1, 2])
Out[7]:
<IntegerArray>
[1, 2]
Length: 2, dtype: Int64
```

**Warning:** Currently `pandas.array()` and `pandas.Series()` use different rules for dtype inference. `pandas.array()` will infer a nullable integer dtype.

```python
In [6]: pd.array([1, None])
Out[6]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64
```

```python
In [7]: pd.array([1, 2])
Out[7]:
<IntegerArray>
[1, 2]
Length: 2, dtype: Int64
```
For backwards-compatibility, `Series` infers these as either integer or float dtype

```
In [8]: pd.Series([1, None])
Out[8]:
0   1.0
1   NaN
dtype: float64
```

```
In [9]: pd.Series([1, 2])
Out[9]:
0   1
1   2
dtype: int64
```

We recommend explicitly providing the dtype to avoid confusion.

```
In [10]: pd.array([1, None], dtype="Int64")
Out[10]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64
```

```
In [11]: pd.Series([1, None], dtype="Int64")
Out[11]:
0   1
1   <NA>
dtype: Int64
```

In the future, we may provide an option for `Series` to infer a nullable-integer dtype.

### 2.13.2 Operations

Operations involving an integer array will behave similar to NumPy arrays. Missing values will be propagated, and the data will be coerced to another dtype if needed.

```
In [12]: s = pd.Series([1, 2, None], dtype="Int64")

# arithmetic
In [13]: s + 1
Out[13]:
0   2
1   3
2   <NA>
dtype: Int64

# comparison
In [14]: s == 1
Out[14]:
0   True
1   False
2   <NA>
dtype: boolean
```

(continues on next page)
# indexing

```python
In[15]: s.iloc[1:3]
Out[15]:
1  2
2 <NA>
dtype: Int64
```

# operate with other dtypes

```python
In[16]: s + s.iloc[1:3].astype("Int8")
Out[16]:
0 <NA>
1  4
2 <NA>
dtype: Int64
```

# coerce when needed

```python
In[17]: s + 0.01
Out[17]:
0  1.01
1  2.01
2 <NA>
dtype: Float64
```

These dtypes can operate as part of DataFrame.

```python
In[18]: df = pd.DataFrame({"A": s, "B": [1, 1, 3], "C": list("aab")})
```

```python
In[19]: df
Out[19]:
    A  B  C
0   1  1  a
1   2  1  a
2 <NA> 3  b
```

```python
In[20]: df.dtypes
Out[20]:
A  Int64
B  int64
C  object
dtype: object
```

These dtypes can be merged & reshaped & casted.

```python
In[21]: pd.concat([df["A"], df["B", "C"]], axis=1).dtypes
Out[21]:
A  Int64
B  int64
C  object
dtype: object
```

```python
In[22]: df["A"].astype(float)
```

(continues on next page)
Reduction and groupby operations such as ‘sum’ work as well.

```python
In [23]: df.sum()
Out[23]:
A 3
B 5
C aab
dtype: object

In [24]: df.groupby("B").A.sum()
Out[24]:
B
1 3
3 0
Name: A, dtype: Int64
```

### 2.13.3 Scalar NA Value

`arrays.IntegerArray` uses `pandas.NA` as its scalar missing value. Slicing a single element that’s missing will return `pandas.NA`.

```python
In [25]: a = pd.array([1, None], dtype="Int64")
In [26]: a[1]
Out[26]: <NA>
```

### 2.14 Nullable Boolean data type

**Note:** BooleanArray is currently experimental. Its API or implementation may change without warning.

New in version 1.0.0.

#### 2.14.1 Indexing with NA values

pandas allows indexing with NA values in a boolean array, which are treated as `False`.

Changed in version 1.0.2.

```python
In [1]: s = pd.Series([1, 2, 3])
In [2]: mask = pd.array([True, False, pd.NA], dtype="boolean")
```

(continues on next page)
If you would prefer to keep the NA values you can manually fill them with `fillna(True)`.

```
In [4]: s[mask].fillna(True)
Out[4]:
0   1
2   3
dtype: int64
```

### 2.14.2 Kleene logical operations

`arrays.BooleanArray` implements Kleene Logic (sometimes called three-value logic) for logical operations like `&` (and), `|` (or) and `^` (exclusive-or).

This table demonstrates the results for every combination. These operations are symmetrical, so flipping the left- and right-hand side makes no difference in the result.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True &amp; True</td>
<td>True</td>
</tr>
<tr>
<td>True &amp; False</td>
<td>False</td>
</tr>
<tr>
<td>True &amp; NA</td>
<td>NA</td>
</tr>
<tr>
<td>False &amp; False</td>
<td>False</td>
</tr>
<tr>
<td>False &amp; NA</td>
<td>False</td>
</tr>
<tr>
<td>NA &amp; NA</td>
<td>NA</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>NA</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>NA</td>
</tr>
<tr>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>True ^ True</td>
<td>False</td>
</tr>
<tr>
<td>True ^ False</td>
<td>True</td>
</tr>
<tr>
<td>True ^ NA</td>
<td>NA</td>
</tr>
<tr>
<td>False ^ False</td>
<td>False</td>
</tr>
<tr>
<td>False ^ NA</td>
<td>NA</td>
</tr>
<tr>
<td>NA ^ NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

When an NA is present in an operation, the output value is NA only if the result cannot be determined solely based on the other input. For example, `True | NA` is True, because both `True | True` and `True | False` are True. In that case, we don’t actually need to consider the value of the NA.

On the other hand, `True & NA` is NA. The result depends on whether the NA really is True or False, since `True & True` is True, but `True & False` is False, so we can't determine the output.

This differs from how `np.nan` behaves in logical operations. `pandas` treated `np.nan` is always false in the output.

In or
2.15 Chart Visualization

This section demonstrates visualization through charting. For information on visualization of tabular data please see the section on *Table Visualization*.

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt
In [2]: plt.close("all")
```

We provide the basics in pandas to easily create decent looking plots. See the ecosystem section for visualization libraries that go beyond the basics documented here.

**Note:** All calls to `np.random` are seeded with 123456.
2.15.1 Basic plotting: plot

We will demonstrate the basics, see the *cookbook* for some advanced strategies.

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`:

```python
In [3]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
In [4]: ts = ts.cumsum()
In [5]: ts.plot();
```

If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above.

On DataFrame, `plot()` is a convenience to plot all of the columns with labels:

```python
In [6]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))
In [7]: df = df.cumsum()
In [8]: plt.figure();
In [9]: df.plot();
```
You can plot one column versus another using the x and y keywords in plot():

```python
In [10]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
In [11]: df3['A'] = pd.Series(list(range(len(df))))
In [12]: df3.plot(x='A', y='B');
```
2.15.2 Other plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to `plot()`, and include:

- `bar` or `barh` for bar plots
- `hist` for histogram
- `box` for boxplot
- `kde` or `density` for density plots
- `area` for area plots
- `scatter` for scatter plots
- `hexbin` for hexagonal bin plots
- `pie` for pie plots

For example, a bar plot can be created the following way:
In [13]: plt.figure();
In [14]: df.iloc[5].plot(kind="bar");

You can also create these other plots using the methods DataFrame.plot.<kind> instead of providing the kind keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

In [15]: df = pd.DataFrame()
In [16]: df.plot.<TAB>  # noqa: E225, E999
   df.plot.area   df.plot.barh   df.plot.density   df.plot.hist   df.plot.line   df.plot.scatter
   df.plot.bar   df.plot.box   df.plot.hexbin   df.plot.kde   df.plot.pie

In addition to these kind s, there are the DataFrame.hist(), and DataFrame.boxplot() methods, which use a separate interface.

Finally, there are several plotting functions in pandas.plotting that take a Series or DataFrame as an argument. These include:

- Scatter Matrix
- Andrews Curves
- Parallel Coordinates
• Lag Plot
• Autocorrelation Plot
• Bootstrap Plot
• RadViz

Plots may also be adorned with *errorbars* or *tables*.

**Bar plots**

For labeled, non-time series data, you may wish to produce a bar plot:

```python
In [17]: plt.figure();
In [18]: df.iloc[5].plot.bar();
In [19]: plt.axhline(0, color="k");
```

Calling a DataFrame’s `plot.bar()` method produces a multiple bar plot:

```python
In [20]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
In [21]: df2.plot.bar();
```
To produce a stacked bar plot, pass `stacked=True`:

```python
In [22]: df2.plot.bar(stacked=True);
```
To get horizontal bar plots, use the `barh` method:

```python
In [23]: df2.plot.barh(stacked=True);
```
Histograms

Histograms can be drawn by using the DataFrame.plot.hist() and Series.plot.hist() methods.

```python
In [24]: df4 = pd.DataFrame(
    ....:     {
    ....:         "a": np.random.randn(1000) + 1,
    ....:         "b": np.random.randn(1000),
    ....:         "c": np.random.randn(1000) - 1,
    ....:     },
    ....:     columns=['a', 'b', 'c'],
    ....: )

In [25]: plt.figure()

In [26]: df4.plot.hist(alpha=0.5);
```
A histogram can be stacked using `stacked=True`. Bin size can be changed using the `bins` keyword.

```
In [27]: plt.figure();
In [28]: df4.plot.hist(stacked=True, bins=20);
```
You can pass other keywords supported by matplotlib `hist`. For example, horizontal and cumulative histograms can be drawn by `orientation='horizontal'` and `cumulative=True`.

```
In [29]: plt.figure();

In [30]: df4["a"].plot.hist(orientation="horizontal", cumulative=True);
```
See the `hist` method and the `matplotlib hist documentation` for more.

The existing interface `DataFrame.hist` to plot histogram still can be used.

```python
In [31]: plt.figure();
In [32]: df['A'].diff().hist();
```
DataFrame.hist() plots the histograms of the columns on multiple subplots:

```
In [33]: plt.figure();
In [34]: df.diff().hist(color="k", alpha=0.5, bins=50);
```
The by keyword can be specified to plot grouped histograms:

```
In [35]: data = pd.Series(np.random.randn(1000))

In [36]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4));
```
In addition, the by keyword can also be specified in DataFrame.plot.hist().

Changed in version 1.4.0.

```
In [37]: data = pd.DataFrame(
           ....:     { 
           ....:         "a": np.random.choice(["x", "y", "z"], 1000), 
           ....:         "b": np.random.choice(["e", "f", "g"], 1000), 
           ....:         "c": np.random.randn(1000), 
           ....:         "d": np.random.randn(1000) - 1, 
           ....:     }, 
           ....: )
```

```
In [38]: data.plot.hist(by=["a", "b"], figsize=(10, 5));
```

2.15. Chart Visualization 691
Box plots

Boxplot can be drawn calling `Series.plot.box()` and `DataFrame.plot.box()`, or `DataFrame.boxplot()` to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

In [39]: df = pd.DataFrame(np.random.rand(10, 5), columns=["A", "B", "C", "D", "E"]) In [40]: df.plot.box();
Boxplot can be colorized by passing `color` keyword. You can pass a dict whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the `dict`, default colors are used for the corresponding artists. Also, boxplot has `sym` keyword to specify fliers style.

When you pass other type of arguments via `color` keyword, it will be directly passed to matplotlib for all the `boxes`, `whiskers`, `medians` and `caps` colorization.

The colors are applied to every boxes to be drawn. If you want more complicated colorization, you can get each drawn artists by passing `return_type`.

```python
In [41]: color = {
   ....:     "boxes": "DarkGreen",
   ....:     "whiskers": "DarkOrange",
   ....:     "medians": "DarkBlue",
   ....:     "caps": "Gray",
   ....: }
   ....:

In [42]: df.plot.box(color=color, sym="r+");
```
Also, you can pass other keywords supported by matplotlib boxplot. For example, horizontal and custom-positioned boxplot can be drawn by `vert=False` and `positions` keywords.

```
In [43]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8]);
```
See the boxplot method and the matplotlib boxplot documentation for more.
The existing interface DataFrame.boxplot to plot boxplot still can be used.

```
In [44]: df = pd.DataFrame(np.random.rand(10, 5))
In [45]: plt.figure();
In [46]: bp = df.boxplot()
```
You can create a stratified boxplot using the by keyword argument to create groupings. For instance,

```python
In [47]: df = pd.DataFrame(np.random.rand(10, 2), columns=["Col1", "Col2"])
In [49]: plt.figure();
In [50]: bp = df.boxplot(by="X")
```
You can also pass a subset of columns to plot, as well as group by multiple columns:

```python
In [51]: df = pd.DataFrame(np.random.rand(10, 3), columns=['Col1', 'Col2', 'Col3'])
In [54]: plt.figure();
In [55]: bp = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```
You could also create groupings with `DataFrame.plot.box()`, for instance:

Changed in version 1.4.0.

```
In [56]: df = pd.DataFrame(np.random.rand(10, 3), columns=['Col1', 'Col2', 'Col3'])
In [58]: plt.figure();
In [59]: bp = df.plot.box(column=['Col1', 'Col2'], by='X')
```
In boxplot, the return type can be controlled by the `return_type` keyword. The valid choices are {"axes", "dict", "both", None}. Faceting, created by DataFrame.boxplot with the by keyword, will affect the output type as well:

<table>
<thead>
<tr>
<th>return_type</th>
<th>Faceted</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No</td>
<td>axes</td>
</tr>
<tr>
<td>None</td>
<td>Yes</td>
<td>2-D ndarray of axes</td>
</tr>
<tr>
<td>'axes'</td>
<td>No</td>
<td>axes</td>
</tr>
<tr>
<td>'axes'</td>
<td>Yes</td>
<td>Series of axes</td>
</tr>
<tr>
<td>'dict'</td>
<td>No</td>
<td>dict of artists</td>
</tr>
<tr>
<td>'dict'</td>
<td>Yes</td>
<td>Series of dicts of artists</td>
</tr>
<tr>
<td>'both'</td>
<td>No</td>
<td>namedtuple</td>
</tr>
<tr>
<td>'both'</td>
<td>Yes</td>
<td>Series of namedtuples</td>
</tr>
</tbody>
</table>

Groupby.boxplot always returns a Series of return_type.

In [60]: np.random.seed(1234)
In [61]: df_box = pd.DataFrame(np.random.randn(50, 2))
In [62]: df_box["g"] = np.random.choice(["A", "B"], size=50)
In [63]: df_box.loc[df_box["g"] == "B", 1] += 3

(continues on next page)
In [64]: bp = df_box.boxplot(by="g")

The subplots above are split by the numeric columns first, then the value of the g column. Below the subplots are first split by the value of g, then by the numeric columns.

In [65]: bp = df_box.groupby("g").boxplot()
Area plot

You can create area plots with `Series.plot.area()` and `DataFrame.plot.area()`. Area plots are stacked by default. To produce stacked area plot, each column must be either all positive or all negative values.

When input data contains NaN, it will be automatically filled by 0. If you want to drop or fill by different values, use `dataframe.dropna()` or `dataframe.fillna()` before calling `plot`.

```
In [66]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [67]: df.plot.area();
```
To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

```python
In [68]: df.plot.area(stacked=False);
```
Scatter plot

Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for the x and y axes. These can be specified by the `x` and `y` keywords.

```
In [69]: df = pd.DataFrame(np.random.rand(50, 4), columns=["a", "b", "c", "d"])

In [70]: df["species"] = pd.Categorical(
    ....:     ["setosa"] * 20 + ["versicolor"] * 20 + ["virginica"] * 10
    ....: )

In [71]: df.plot.scatter(x="a", y="b");
```
To plot multiple column groups in a single axes, repeat `plot` method specifying target `ax`. It is recommended to specify `color` and `label` keywords to distinguish each groups.

```
In [72]: ax = df.plot.scatter(x="a", y="b", color="DarkBlue", label="Group 1")
```

```
In [73]: df.plot.scatter(x="c", y="d", color="DarkGreen", label="Group 2", ax=ax);
```
The keyword `c` may be given as the name of a column to provide colors for each point:

```python
In [74]: df.plot.scatter(x="a", y="b", c="c", s=50);
```
If a categorical column is passed to `c`, then a discrete colorbar will be produced:

New in version 1.3.0.

In [75]: df.plot.scatter(x="a", y="b", c="species", cmap="viridis", s=50);
You can pass other keywords supported by matplotlib `scatter`. The example below shows a bubble chart using a column of the DataFrame as the bubble size.

```python
In [76]: df.plot.scatter(x="a", y="b", s=df["c"] * 200);
```
Hexagonal bin plot

You can create hexagonal bin plots with DataFrame.plot.hexbin(). Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

```python
In [77]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [78]: df['b'] = df['b'] + np.arange(1000)
In [79]: df.plot.hexbin(x='a', y='b', gridsize=25);
```

See the scatter method and the matplotlib scatter documentation for more.
A useful keyword argument is `gridsize`; it controls the number of hexagons in the x-direction, and defaults to 100. A larger `gridsize` means more, smaller bins.

By default, a histogram of the counts around each (x, y) point is computed. You can specify alternative aggregations by passing values to the `C` and `reduce_C_function` arguments. `C` specifies the value at each (x, y) point and `reduce_C_function` is a function of one argument that reduces all the values in a bin to a single number (e.g., `mean`, `max`, `sum`, `std`). In this example the positions are given by columns `a` and `b`, while the value is given by column `z`. The bins are aggregated with NumPy’s `max` function.

```python
In [80]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])
In [81]: df['b'] = df['b'] + np.arange(1000)
In [82]: df['z'] = np.random.uniform(0, 3, 1000)
In [83]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max, gridsize=25);
```
You can create a pie plot with DataFrame.plot.pie() or Series.plot.pie(). If your data includes any NaN, they will be automatically filled with 0. A ValueError will be raised if there are any negative values in your data.

```
In [84]: series = pd.Series(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], name='series')
In [85]: series.plot.pie(figsize=(6, 6));
```

See the hexbin method and the matplotlib hexbin documentation for more.

**Pie plot**

You can create a pie plot with DataFrame.plot.pie() or Series.plot.pie(). If your data includes any NaN, they will be automatically filled with 0. A ValueError will be raised if there are any negative values in your data.
For pie plots it’s best to use square figures, i.e. a figure aspect ratio 1. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling `ax.set_aspect('equal')` on the returned axes object.

Note that pie plot with `DataFrame` requires that you either specify a target column by the `y` argument or `subplots=True`. When `y` is specified, pie plot of selected column will be drawn. If `subplots=True` is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify `legend=False` to hide it.

```python
In [86]: df = pd.DataFrame(
   ....:    3 * np.random.randn(4, 2), index=\"a", \"b", \"c", \"d\", columns=\"x", \"y\"
   ....: )
   ....:
In [87]: df.plot.pie(subplots=True, figsize=(8, 4));
```
You can use the `labels` and `colors` keywords to specify the labels and colors of each wedge.

**Warning:** Most pandas plots use the `label` and `color` arguments (note the lack of “s” on those). To be consistent with `matplotlib.pyplot.pie()` you must use `labels` and `colors`.

If you want to hide wedge labels, specify `labels=None`. If `fontsize` is specified, the value will be applied to wedge labels. Also, other keywords supported by `matplotlib.pyplot.pie()` can be used.

```python
In [88]: series.plot.pie(
    ....:     labels=['AA', 'BB', 'CC', 'DD'],
    ....:     colors=['r', 'g', 'b', 'c'],
    ....:     autopct='%.2f',
    ....:     fontsize=20,
    ....:     figsize=(6, 6),
    ....: );
```

If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```python
In [89]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name="series2")
In [90]: series.plot.pie(figsize=(6, 6));
```
See the matplotlib pie documentation for more.

**2.15.3 Plotting with missing data**

pandas tries to be pragmatic about plotting DataFrames or Series that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.
If any of these defaults are not what you want, or if you want to be explicit about how missing values are handled, consider using `fillna()` or `dropna()` before plotting.

### 2.15.4 Plotting tools

These functions can be imported from `pandas.plotting` and take a `Series` or `DataFrame` as an argument.

#### Scatter matrix plot

You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.plotting`:

```python
In [91]: from pandas.plotting import scatter_matrix
In [92]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])
In [93]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde');```
Density plot

You can create density plots using the `Series.plot.kde()` and `DataFrame.plot.kde()` methods.

```
In [94]: ser = pd.Series(np.random.randn(1000))

In [95]: ser.plot.kde();
```
Andrews curves

Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series, see the Wikipedia entry for more information. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

Note: The “Iris” dataset is available here.

In [96]: from pandas.plotting import andrews_curves

In [97]: data = pd.read_csv("data/iris.data")

In [98]: plt.figure();

In [99]: andrews_curves(data, "Name");
Parallel coordinates

Parallel coordinates is a plotting technique for plotting multivariate data, see the Wikipedia entry for an introduction. Parallel coordinates allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

```
In [100]: from pandas.plotting import parallel_coordinates

In [101]: data = pd.read_csv("data/iris.data")

In [102]: plt.figure();

In [103]: parallel_coordinates(data, "Name");
```
Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random. The `lag` argument may be passed, and when `lag=1` the plot is essentially `data[:-1]` vs. `data[1:]`.  

```
In [104]: from pandas.plotting import lag_plot

In [105]: plt.figure();

In [106]: spacing = np.linspace(-99 * np.pi, 99 * np.pi, num=1000)

In [107]: data = pd.Series(0.1 * np.random.rand(1000) + 0.9 * np.sin(spacing))

In [108]: lag_plot(data);
```
Autocorrelation plot

Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band. See the Wikipedia entry for more about autocorrelation plots.

In [109]: from pandas.plotting import autocorrelation_plot

In [110]: plt.figure();

In [111]: spacing = np.linspace(-9 * np.pi, 9 * np.pi, num=1000)

In [112]: data = pd.Series(0.7 * np.random.rand(1000) + 0.3 * np.sin(spacing))

In [113]: autocorrelation_plot(data);
Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc. A random subset of a specified size is selected from a data set, the statistic in question is computed for this subset and the process is repeated a specified number of times. Resulting plots and histograms are what constitutes the bootstrap plot.

```python
In [114]: from pandas.plotting import bootstrap_plot

In [115]: data = pd.Series(np.random.rand(1000))

In [116]: bootstrap_plot(data, size=50, samples=500, color="grey");
```
RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs it will be colored differently. See the R package Radviz for more information.

Note: The “Iris” dataset is available here.

```python
In [117]: from pandas.plotting import radviz

In [118]: data = pd.read_csv("data/iris.data")

In [119]: plt.figure();

In [120]: radviz(data, "Name");
```
2.15.5 Plot formatting

Setting the plot style

From version 1.5 and up, matplotlib offers a range of pre-configured plotting styles. Setting the style can be used to easily give plots the general look that you want. Setting the style is as easy as calling `matplotlib.style.use(my_plot_style)` before creating your plot. For example you could write `matplotlib.style.use('ggplot')` for ggplot-style plots.

You can see the various available style names at `matplotlib.style.available` and it’s very easy to try them out.

General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [121]: plt.figure();
In [122]: ts.plot(style="k--", label="Series");
```
For each kind of plot (e.g. line, bar, scatter) any additional arguments keywords are passed along to the corresponding matplotlib function (ax.plot(), ax.bar(), ax.scatter()). These can be used to control additional styling, beyond what pandas provides.

**Controlling the legend**

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```python
In [123]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))

In [124]: df = df.cumsum()

In [125]: df.plot(legend=False);
```
Controlling the labels

New in version 1.1.0.

You may set the xlabel and ylabel arguments to give the plot custom labels for x and y axis. By default, pandas will pick up index name as xlabel, while leaving it empty for ylabel.

```
In [126]: df.plot();
```

```
In [127]: df.plot(xlabel="new x", ylabel="new y");
```
You may pass `logy` to get a log-scale Y axis.

```python
In [128]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

In [129]: ts = np.exp(ts.cumsum())

In [130]: ts.plot(logy=True);
```
Plotting on a secondary y-axis

To plot data on a secondary y-axis, use the secondary_y keyword:

```
In [131]: df["A"].plot();
In [132]: df["B"].plot(secondary_y=True, style="g");
```
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```python
In [133]: plt.figure();
In [134]: ax = df.plot(secondary_y=['A', 'B'])
In [135]: ax.set_ylabel("CD scale");
In [136]: ax.right_ax.set_ylabel("AB scale");
```
Note that the columns plotted on the secondary y-axis is automatically marked with “(right)” in the legend. To turn off the automatic marking, use the `mark_right=False` keyword:

```python
In [137]: plt.figure();
In [138]: df.plot(secondary_y=['A', 'B'], mark_right=False);
```
Custom formatters for timeseries plots

Changed in version 1.0.0.

pandas provides custom formatters for timeseries plots. These change the formatting of the axis labels for dates and times. By default, the custom formatters are applied only to plots created by pandas with DataFrame.plot() or Series.plot(). To have them apply to all plots, including those made by matplotlib, set the option pd.options.plotting.matplotlib.register_converters = True or use pandas.plotting.register_matplotlib_converters().

Suppressing tick resolution adjustment

pandas includes automatic tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created twinx), you can choose to suppress this behavior for alignment purposes.

Here is the default behavior, notice how the x-axis tick labeling is performed:

```python
In [139]: plt.figure();
In [140]: df['A'].plot();
```
Using the `x_compat` parameter, you can suppress this behavior:

```
In [141]: plt.figure();

In [142]: df['A'].plot(x_compat=True);
```
If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plotting.plot_params` can be used in a `with` statement:

```
In [143]: plt.figure();

In [144]: with pd.plotting.plot_params.use("x_compat", True):
     ....:     df["A"].plot(color="r")
     ....:     df["B"].plot(color="g")
     ....:     df["C"].plot(color="b")
     ....:
```
Automatic date tick adjustment

TimedeltaIndex now uses the native matplotlib tick locator methods, it is useful to call the automatic date tick adjustment from matplotlib for figures whose ticklabels overlap.

See the autofmt_xdate method and the matplotlib documentation for more.
Subplots

Each Series in a DataFrame can be plotted on a different axis with the subplots keyword:

```
In [145]: df.plot(subplots=True, figsize=(6, 6));
```
Using layout and targeting multiple axes

The layout of subplots can be specified by the `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If the input is invalid, a `ValueError` will be raised.

The number of axes which can be contained by rows x columns specified by `layout` must be larger than the number of required subplots. If layout can contain more axes than required, blank axes are not drawn. Similar to a NumPy array’s `reshape` method, you can use `-1` for one dimension to automatically calculate the number of rows or columns needed, given the other.

```python
In [146]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```

The above example is identical to using:

```python
In [147]: df.plot(subplots=True, layout=(2, -1), figsize=(6, 6), sharex=False);
```

The required number of columns (3) is inferred from the number of series to plot and the given number of rows (2).
You can pass multiple axes created beforehand as list-like via `ax` keyword. This allows more complicated layouts. The passed axes must be the same number as the subplots being drawn.

When multiple axes are passed via the `ax` keyword, `layout`, `sharex` and `sharey` keywords don’t affect to the output. You should explicitly pass `sharex=False` and `sharey=False`, otherwise you will see a warning.

```python
In [148]: fig, axes = plt.subplots(4, 4, figsize=(9, 9))
In [149]: plt.subplots_adjust(wspace=0.5, hspace=0.5)
In [150]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
In [151]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]
In [152]: df.plot(subplots=True, ax=target1, legend=False, sharex=False, sharey=False);
In [153]: (-df).plot(subplots=True, ax=target2, legend=False, sharex=False, sharey=False);
```
Another option is passing an `ax` argument to `Series.plot()` to plot on a particular axis:

```
In [154]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [155]: plt.subplots_adjust(wspace=0.2, hspace=0.5)

In [156]: df['A'].plot(ax=axes[0, 0]);

In [157]: axes[0, 0].set_title("A");

In [158]: df['B'].plot(ax=axes[0, 1]);

In [159]: axes[0, 1].set_title("B");
```

(continues on next page)
Plotting with error bars

Plotting with error bars is supported in `DataFrame.plot()` and `Series.plot()`.

Horizontal and vertical error bars can be supplied to the `xerr` and `yerr` keyword arguments to `plot()`. The error values can be specified using a variety of formats:

- As a `DataFrame` or `dict` of errors with column names matching the `columns` attribute of the plotting `DataFrame` or matching the `name` attribute of the `Series`.
- As a `str` indicating which of the columns of plotting `DataFrame` contain the error values.
- As raw values (list, tuple, or `np.ndarray`). Must be the same length as the plotting `DataFrame/Series`.

Here is an example of one way to easily plot group means with standard deviations from the raw data.
# Generate the data

In [164]: ix3 = pd.MultiIndex.from_arrays(
    ....:     [  
    ....:         ["a", "a", "a", "a", "b", "b", "b", "b"],  
    ....:         ["foo", "foo", "foo", "bar", "bar", "foo", "bar", "bar"],  
    ....:     ],  
    ....:     names=["letter", "word"],  
    ....: )

In [165]: df3 = pd.DataFrame(
    ....:     {  
    ....:         "data1": [9, 3, 2, 4, 3, 2, 4, 6, 3, 2],  
    ....:         "data2": [9, 6, 5, 7, 5, 4, 5, 6, 5, 1],  
    ....:     },  
    ....:     index=ix3,
    ....: )

# Group by index labels and take the means and standard deviations # for each group

In [166]: gp3 = df3.groupby(level=("letter", "word"))

In [167]: means = gp3.mean()

In [168]: errors = gp3.std()

In [169]: means

Out[169]:

    data1    data2
    letter word
    a   bar 3.500000 6.000000
         foo 4.666667 6.666667
    b   bar 3.666667 4.000000
         foo 3.000000 4.500000

In [170]: errors

Out[170]:

    data1    data2
    letter word
    a   bar 0.707107 1.414214
         foo 3.785939 2.081666
    b   bar 2.081666 2.645751
         foo 1.414214 0.707107

# Plot

In [171]: fig, ax = plt.subplots()

In [172]: means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a $N$ length Series, a $2xN$ array should be provided indicating lower and upper (or left and right) errors. For a $MxN$ DataFrame, asymmetrical errors should be in a $Mx2xN$ array.

Here is an example of one way to plot the min/max range using asymmetrical error bars.

```python
In [173]: mins = gp3.min()
In [174]: maxs = gp3.max()

# errors should be positive, and defined in the order of lower, upper
In [175]: errors = [[means[c] - mins[c], maxs[c] - means[c]] for c in df3.columns]

# Plot
In [176]: fig, ax = plt.subplots()

In [177]: means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```
Plotting tables

Plotting with matplotlib table is now supported in DataFrame.plot() and Series.plot() with a table keyword. The table keyword can accept bool, DataFrame or Series. The simple way to draw a table is to specify table=True. Data will be transposed to meet matplotlib’s default layout.

```python
In [178]: fig, ax = plt.subplots(1, 1, figsize=(7, 6.5))

In [179]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

In [180]: ax.xaxis.tick_top()  # Display x-axis ticks on top.

In [181]: df.plot(table=True, ax=ax);
```
Also, you can pass a different DataFrame or Series to the `table` keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as seen in the example below.

```python
In [182]: fig, ax = plt.subplots(1, 1, figsize=(7, 6.75))

In [183]: ax.xaxis.tick_top()  # Display x-axis ticks on top.

In [184]: df.plot(table=np.round(df.T, 2), ax=ax);
```
There also exists a helper function `pandas.plotting.table`, which creates a table from `DataFrame` or `Series`, and adds it to a `matplotlib.Axes` instance. This function can accept keywords which the `matplotlib` `table` has.

In [185]: from pandas.plotting import table

In [186]: fig, ax = plt.subplots(1, 1)

In [187]: table(ax, np.round(df.describe(), 2), loc="upper right", colWidths=[0.2, 0.2, 0.2]);

In [188]: df.plot(ax=ax, ylim=(0, 2), legend=None);
Note: You can get table instances on the axes using `axes.tables` property for further decorations. See the matplotlib table documentation for more.

Colormaps

A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, DataFrame plotting supports the use of the `colormap` argument, which accepts either a Matplotlib colormap or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available here.

As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the DataFrame. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

To use the cubehelix colormap, we can pass `colormap='cubehelix'`.

```python
In [189]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
In [190]: df = df.cumsum()
In [191]: plt.figure();
In [192]: df.plot(colormap="cubehelix");
```
Alternatively, we can pass the colormap itself:

```
In [193]: from matplotlib import cm

In [194]: plt.figure();

In [195]: df.plot(colormap=cm.cubehelix);
```
Colormaps can also be used other plot types, like bar charts:

```python
In [196]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
In [197]: dd = dd.cumsum()
In [198]: plt.figure();
In [199]: dd.plot.bar(colormap="Greens");
```
Parallel coordinates charts:

In [200]: plt.figure();

In [201]: parallel_coordinates(data, "Name", colormap="gist_rainbow");
Andrews curves charts:

```python
In [202]: plt.figure();

In [203]: andrews_curves(data, "Name", colormap="winter");
```
2.15.6 Plotting directly with matplotlib

In some situations it may still be preferable or necessary to prepare plots directly with matplotlib, for instance when a certain type of plot or customization is not (yet) supported by pandas. Series and DataFrame objects behave like arrays and can therefore be passed directly to matplotlib functions without explicit casts.

pandas also automatically registers formatters and locators that recognize date indices, thereby extending date and time support to practically all plot types available in matplotlib. Although this formatting does not provide the same level of refinement you would get when plotting via pandas, it can be faster when plotting a large number of points.

```python
In [204]: price = pd.Series(
       ....:     np.random.randn(150).cumsum(),
       ....:     index=pd.date_range("2000-1-1", periods=150, freq="B"),
       ....:     )
       ....:
In [205]: ma = price.rolling(20).mean()
In [206]: mstd = price.rolling(20).std()
In [207]: plt.figure();
```

(continues on next page)
2.15.7 Plotting backends

Starting in version 0.25, pandas can be extended with third-party plotting backends. The main idea is letting users select a plotting backend different than the provided one based on Matplotlib.

This can be done by passing ‘backend.module’ as the argument backend in plot function. For example:

```python
>>> Series([1, 2, 3]).plot(backend="backend.module")
```

Alternatively, you can also set this option globally, do you don’t need to specify the keyword in each `plot` call. For example:

```python
>>> pd.set_option("plotting.backend", "backend.module")
>>> pd.Series([1, 2, 3]).plot()
```
Or:

```python
>>> pd.options.plotting.backend = "backend.module"
>>> pd.Series([1, 2, 3]).plot()
```

This would be more or less equivalent to:

```python
>>> import backend.module
>>> backend.module.plot(pd.Series([1, 2, 3]))
```

The backend module can then use other visualization tools (Bokeh, Altair, hvplot...) to generate the plots. Some libraries implementing a backend for pandas are listed on the ecosystem ecosystem.visualization page.

Developers guide can be found at https://pandas.pydata.org/docs/dev/development/extending.html#plotting-backends

### 2.16 Table Visualization

This section demonstrates visualization of tabular data using the **Styler** class. For information on visualization with charting please see **Chart Visualization**. This document is written as a Jupyter Notebook, and can be viewed or downloaded here.

#### 2.16.1 Styler Object and HTML

Styling should be performed after the data in a DataFrame has been processed. The **Styler** creates an HTML `<table>` and leverages CSS styling language to manipulate many parameters including colors, fonts, borders, background, etc. See here for more information on styling HTML tables. This allows a lot of flexibility out of the box, and even enables web developers to integrate DataFrames into their exiting user interface designs.

The DataFrame.style attribute is a property that returns a **Styler** object. It has a _repr_html_ method defined on it so they are rendered automatically in Jupyter Notebook.

```python
[2]: import pandas as pd
import numpy as np
import matplotlib as mpl

df = pd.DataFrame([[38.0, 2.0, 18.0, 22.0, 21, np.nan],
                   [19, 439, 6, 452, 226,232]],
                  index=pd.Index(['Tumour (Positive)', 'Non-Tumour (Negative)'],
                                 name='Actual Label:'),
                  columns=pd.MultiIndex.from_product([[Decision Tree', 'Regression',
                                 'Random'],[Tumour', 'Non-Tumour'],
                                 name=[Model:', 'Predicted:'])))

df.style
```

The above output looks very similar to the standard DataFrame HTML representation. But the HTML here has already attached some CSS classes to each cell, even if we haven’t yet created any styles. We can view these by calling the .to_html() method, which returns the raw HTML as string, which is useful for further processing or adding to a file - read on in More about CSS and HTML. Below we will show how we can use these to format the DataFrame to be more communicative. For example how we can build s:

```python
[4]: s
```

```python
[4]: <pandas.io.format.style.Styler at 0x7f9e346e4a00>
```
2.16.2 Formatting the Display

Formatting Values

Before adding styles it is useful to show that the Styler can distinguish the display value from the actual value, in both datavlaues and index or columns headers. To control the display value, the text is printed in each cell as string, and we can use the `format()` and `format_index()` methods to manipulate this according to a format spec string or a callable that takes a single value and returns a string. It is possible to define this for the whole table, or index, or for individual columns, or MultiIndex levels.

Additionally, the format function has a precision argument to specifically help formatting floats, as well as decimal and thousands separators to support other locales, an na_rep argument to display missing data, and an escape argument to help displaying safe-HTML or safe-LaTeX. The default formatter is configured to adopt pandas’ styler.format.precision option, controllable using with `pd.option_context('format.precision', 2):

```python
[5]: df.style.format(precision=0, na_rep='MISSING', thousands=" ",
    formatter={('Decision Tree', 'Tumour'): "{:.2f}",
    ('Regression', 'Non-Tumour'): lambda x: "$ {:.1f}".format(x*-
       ˓→1e6)}
[5]: <pandas.io.formats.style.Styler at 0x7f9e346faac0>
```

Using Styler to manipulate the display is a useful feature because maintaining the indexing and datavalues for other purposes gives greater control. You do not have to overwrite your DataFrame to display it how you like. Here is an example of using the formatting functions whilst still relying on the underlying data for indexing and calculations.

```python
[6]: weather_df = pd.DataFrame(np.random.rand(10,2)*5,
    index=pd.date_range(start="2021-01-01", periods=10),
    columns=["Tokyo", "Beijing"]

def rain_condition(v):
    if v < 1.75:
        return "Dry"
    elif v < 2.75:
        return "Rain"
    return "Heavy Rain"

def make_pretty(styler):
    styler.set_caption("Weather Conditions")
    styler.format(rain_condition)
    styler.format_index(lambda v: v.strftime("%A"))
    styler.background_gradient(axis=None, vmin=1, vmax=5, cmap="YlGnBu")
    return styler

weather_df
```

(continues on next page)
```
2021-01-08  3.250209  1.446595
2021-01-09  0.777491  3.677234
2021-01-10  1.415225  0.526102
```

```
[7]: weather_df.loc["2021-01-04":"2021-01-08"].style.pipe(make_pretty)
[7]: <pandas.io.formats.style.Styler at 0x7f9e346fa3a0>
```

Hiding Data

The index and column headers can be completely hidden, as well subselecting rows or columns that one wishes to exclude. Both these options are performed using the same methods.

The index can be hidden from rendering by calling `.hide()` without any arguments, which might be useful if your index is integer based. Similarly column headers can be hidden by calling `.hide(axis=“columns”)` without any further arguments.

Specific rows or columns can be hidden from rendering by calling the same `.hide()` method and passing in a row/column label, a list-like or a slice of row/column labels to for the `subset` argument.

Hiding does not change the integer arrangement of CSS classes, e.g. hiding the first two columns of a DataFrame means the column class indexing will still start at `col2`, since `col0` and `col1` are simply ignored.

We can update our Styler object from before to hide some data and format the values.

```
[8]: s = df.style.format('{{:.0f}}').hide([('Random', 'Tumour'), ('Random', 'Non-Tumour')], axis=“columns”)
s
[8]: <pandas.io.formats.style.Styler at 0x7f9e346e4df0>
```

2.16.3 Methods to Add Styles

There are 3 primary methods of adding custom CSS styles to Styler:

- Using `.set_table_styles()` to control broader areas of the table with specified internal CSS. Although table styles allow the flexibility to add CSS selectors and properties controlling all individual parts of the table, they are unwieldy for individual cell specifications. Also, note that table styles cannot be exported to Excel.

- Using `.set_td_classes()` to directly link either external CSS classes to your data cells or link the internal CSS classes created by `.set_table_styles()`. See here. These cannot be used on column header rows or indexes, and also won’t export to Excel.

- Using the `.apply()` and `.applymap()` functions to add direct internal CSS to specific data cells. See here. As of v1.4.0 there are also methods that work directly on column header rows or indexes; `.apply_index()` and `.applymap_index()`. Note that only these methods add styles that will export to Excel. These methods work in a similar way to `DataFrame.apply()` and `DataFrame.applymap()`.
2.16.4 Table Styles

Table styles are flexible enough to control all individual parts of the table, including column headers and indexes. However, they can be unwieldy to type for individual data cells or for any kind of conditional formatting, so we recommend that table styles are used for broad styling, such as entire rows or columns at a time.

Table styles are also used to control features which can apply to the whole table at once such as creating a generic hover functionality. The :hover pseudo-selector, as well as other pseudo-selectors, can only be used this way.

To replicate the normal format of CSS selectors and properties (attribute value pairs), e.g.

```css
tr:hover {
  background-color: #ffff99;
}
```

the necessary format to pass styles to `set_table_styles()` is as a list of dicts, each with a CSS-selector tag and CSS-properties. Properties can either be a list of 2-tuples, or a regular CSS-string, for example:

```
[10]: cell_hover = {
    'selector': 'td:hover',
    'props': [('
        background-color', '#ffffb3')]
}
index_names = {
    'selector': '.index_name',
    'props': 'font-style: italic; color: darkgrey; font-weight:normal;'
}
headers = {
    'selector': 'th:not(.index_name)',
    'props': 'background-color: #000066; color: white;'
}
s.set_table_styles([cell_hover, index_names, headers])
```

Next we just add a couple more styling artifacts targeting specific parts of the table. Be careful here, since we are chaining methods we need to explicitly instruct the method not to overwrite the existing styles.

```
[12]: s.set_table_styles([
    {'selector': 'th.col_heading', 'props': 'text-align: center;'},
    {'selector': 'th.col_heading.level0', 'props': 'font-size: 1.5em;'},
    {'selector': 'td', 'props': 'text-align: center; font-weight: bold;'},
], overwrite=False)

[12]: <pandas.io.formats.style.Styler at 0x7f9e346e4df0>
```

As a convenience method (since version 1.2.0) we can also pass a `dict` to `set_table_styles()` which contains row or column keys. Behind the scenes Styler just indexes the keys and adds relevant `.col<m>` or `.row<n>` classes as necessary to the given CSS selectors.

```
[14]: s.set_table_styles({
    ('Regression', 'Tumour'): [{
        'selector': 'th', 'props': 'border-left: 1px solid white'},
        {'selector': 'td', 'props': 'border-left: 1px solid #000066'}
    }, overwrite=False, axis=0)
```

754 Chapter 2. User Guide
2.16.5 Setting Classes and Linking to External CSS

If you have designed a website then it is likely you will already have an external CSS file that controls the styling of table and cell objects within it. You may want to use these native files rather than duplicate all the CSS in python (and duplicate any maintenance work).

**Table Attributes**

It is very easy to add a `class` to the main `table` using `.set_table_attributes()`. This method can also attach inline styles - read more in CSS Hierarchies.

```python
out = s.set_table_attributes('class="my-table-cls"').to_html()
print(out[out.find('<table'):][109:]

<table id="T_xyz01" class="my-table-cls">
<thead>
<tr>
<th class="index_name level0" >Model:</th>
</tr>
</thead>
</table>
```

**Data Cell CSS Classes**

*New in version 1.2.0*

The `.set_td_classes()` method accepts a DataFrame with matching indices and columns to the underlying Styler’s DataFrame. That DataFrame will contain strings as css-classes to add to individual data cells: the `<td>` elements of the `<table>`. Rather than use external CSS we will create our classes internally and add them to table style. We will save adding the borders until the section on tooltips.

```python
s.set_table_styles([  # create internal CSS classes
    {'selector': '.true', 'props': 'background-color: #e6ffe6;'},
    {'selector': '.false', 'props': 'background-color: #ffe6e6;'},
], overwrite=False)
cell_color = pd.DataFrame([[true, false, true, false],
                           [false, true, false, true]],
                           index=df.index,
                           columns=df.columns[:4])
s.set_td_classes(cell_color)
```
2.16.6 Styler Functions

Acting on Data

We use the following methods to pass your style functions. Both of those methods take a function (and some other keyword arguments) and apply it to the DataFrame in a certain way, rendering CSS styles.

- `.applymap()` (elementwise): accepts a function that takes a single value and returns a string with the CSS attribute-value pair.
- `.apply()` (column-/row-/table-wise): accepts a function that takes a Series or DataFrame and returns a Series, DataFrame, or numpy array with an identical shape where each element is a string with a CSS attribute-value pair. This method passes each column or row of your DataFrame one-at-a-time or the entire table at once, depending on the axis keyword argument. For columnwise use axis=0, rowwise use axis=1, and for the entire table at once use axis=None.

This method is powerful for applying multiple, complex logic to data cells. We create a new DataFrame to demonstrate this.

```python
[19]: np.random.seed(0)
df2 = pd.DataFrame(np.random.randn(10,4), columns=['A','B','C','D'])
df2.style
[19]: <pandas.io.formats.style.Styler at 0x7f9e28628310>
```

For example we can build a function that colors text if it is negative, and chain this with a function that partially fades cells of negligible value. Since this looks at each element in turn we use applymap.

```python
[20]: def style_negative(v, props=''):  
    return props if v < 0 else None
s2 = df2.style.applymap(style_negative, props='color:red;')\ 
    .applymap(lambda v: 'opacity: 20%;' if (v < 0.3) and (v > -0.3) else None)
s2
[20]: <pandas.io.formats.style.Styler at 0x7f9e34542d00>
```

We can also build a function that highlights the maximum value across rows, cols, and the DataFrame all at once. In this case we use apply. Below we highlight the maximum in a column.

```python
[22]: def highlight_max(s, props=''):  
    return np.where(s == np.nanmax(s.values), props, '')
    s2.apply(highlight_max, props='color:white;background-color:darkblue;', axis=0)
[22]: <pandas.io.formats.style.Styler at 0x7f9e34542d00>
```

We can use the same function across the different axes, highlighting here the DataFrame maximum in purple, and row maximums in pink.

```python
[24]: s2.apply(highlight_max, props='color:white;background-color:pink;', axis=1)\ 
    .apply(highlight_max, props='color:white;background-color:purple', axis=None)
[24]: <pandas.io.formats.style.Styler at 0x7f9e34542d00>
```

This last example shows how some styles have been overwritten by others. In general the most recent style applied is active but you can read more in the section on CSS hierarchies. You can also apply these styles to more granular parts of the DataFrame - read more in section on subset slicing.

It is possible to replicate some of this functionality using just classes but it can be more cumbersome. See item 3) of Optimization
Debugging Tip: If you’re having trouble writing your style function, try just passing it into DataFrame.apply. Internally, Styler.apply uses DataFrame.apply so the result should be the same, and with DataFrame.apply you will be able to inspect the CSS string output of your intended function in each cell.

Acting on the Index and Column Headers

Similar application is achieved for headers by using:

- **.applymap_index()** (elementwise): accepts a function that takes a single value and returns a string with the CSS attribute-value pair.
- **.apply_index()** (level-wise): accepts a function that takes a Series and returns a Series, or numpy array with an identical shape where each element is a string with a CSS attribute-value pair. This method passes each level of your Index one-at-a-time. To style the index use `axis=0` and to style the column headers use `axis=1`.

You can select a level of a MultiIndex but currently no similar subset application is available for these methods.

```python
s2.applymap_index(lambda v: "color:pink;" if v>4 else "color:darkblue;", axis=0)
s2.apply_index(lambda s: np.where(s.isin(['A', 'B'])), "color:pink;", "color:darkblue;"), axis=1)
```

2.16.7 Tooltips and Captions

Table captions can be added with the **.set_caption()** method. You can use table styles to control the CSS relevant to the caption.

```python
s.set_caption("Confusion matrix for multiple cancer prediction models.")
.s.set_table_styles([{
    'selector': 'caption',
    'props': 'caption-side: bottom; font-size:1.25em;' 
}])
```

Adding tooltips (since version 1.3.0) can be done using the **.set_tooltips()** method in the same way you can add CSS classes to data cells by providing a string based DataFrame with intersecting indices and columns. You don't have to specify a css_class name or any css props for the tooltips, since there are standard defaults, but the option is there if you want more visual control.

```python
tt = pd.DataFrame([['This model has a very strong true positive rate',
    'This model's total number of false negatives is too high']],
    index=['Tumour (Positive)'], columns=df.columns[[0,3]])
s.set_tooltips(tt, props="visibility: hidden; position: absolute; z-index: 1; border-\n"1px solid #000066;'
    'background-color: white; color: #000066; font-size: 0.8em;'
    'transform: translate(0px, -24px); padding: 0.6em; border-\n"radius: 0.5em;"
```

2.16. Table Visualization
The only thing left to do for our table is to add the highlighting borders to draw the audience attention to the tooltips. We will create internal CSS classes as before using table styles. Setting classes always overwrites so we need to make sure we add the previous classes.

```python
s.set_table_styles([  
    # create internal CSS classes
    {'selector': '.border-red', 'props': 'border: 2px dashed red;'},
    {'selector': '.border-green', 'props': 'border: 2px dashed green;'},
], overwrite=False)
```

```python
cell_border = pd.DataFrame(
    [['border-green', '', '', ''],
     ['', '', '', '']],
    index=df.index,
    columns=df.columns[:4])
```

```python
s.set_td_classes(cell_color + cell_border)
```

2.16.8 Finer Control with Slicing

The examples we have shown so far for the Styler.apply and Styler.applymap functions have not demonstrated the use of the subset argument. This is a useful argument which permits a lot of flexibility: it allows you to apply styles to specific rows or columns, without having to code that logic into your style function.

The value passed to subset behaves similar to slicing a DataFrame:

- A scalar is treated as a column label
- A list (or Series or NumPy array) is treated as multiple column labels
- A tuple is treated as (row_indexer, column_indexer)

Consider using pd.IndexSlice to construct the tuple for the last one. We will create a MultiIndexed DataFrame to demonstrate the functionality.

```python
df3 = pd.DataFrame(np.random.randn(4,4),
                  pd.MultiIndex.from_product([['A', 'B'], ['r1', 'r2']]),
                  columns=['c1','c2','c3','c4'])
```

We will use subset to highlight the maximum in the third and fourth columns with red text. We will highlight the subset sliced region in yellow.

```python
slice_ = ['c3', 'c4']
df3.style.apply(highlight_max, props='color:red;', axis=0, subset=slice_)
   .set_properties(**{  
       'background-color': '#ffffb3',
     }, subset=slice_)
```

If combined with the IndexSlice as suggested then it can index across both dimensions with greater flexibility.
This also provides the flexibility to sub select rows when used with the axis=1.

There is also scope to provide conditional filtering.

Suppose we want to highlight the maximum across columns 2 and 4 only in the case that the sum of columns 1 and 3 is less than -2.0 (essentially excluding rows [:, 'r2']).

Only label-based slicing is supported right now, not positional, and not callables.

If your style function uses a subset or axis keyword argument, consider wrapping your function in a `functools.partial`, partialing out that keyword.

2.16.9 Optimization

Generally, for smaller tables and most cases, the rendered HTML does not need to be optimized, and we don’t really recommend it. There are two cases where it is worth considering:

- If you are rendering and styling a very large HTML table, certain browsers have performance issues.
- If you are using Styler to dynamically create part of online user interfaces and want to improve network performance.

Here we recommend the following steps to implement:

1. Remove UUID and cell_ids

Ignore the uuid and set cell_ids to False. This will prevent unnecessary HTML.
This is better:

```python
from pandas.io.formats.style import Styler
s4 = Styler(df4, uuid_len=0, cell_ids=False)
```

2. Use table styles

Use table styles where possible (e.g. for all cells or rows or columns at a time) since the CSS is nearly always more efficient than other formats.

This is sub-optimal:

```python
props = 'font-family: "Times New Roman", Times, serif; color: #e83e8c; font-size:1.3em;
df4.style.applymap(lambda x: props, subset=[1])
```

This is better:

```python
df4.style.set_table_styles([{
'selector': 'td.col1', 'props': props}
])
```

3. Set classes instead of using Styler functions

For large DataFrames where the same style is applied to many cells it can be more efficient to declare the styles as classes and then apply those classes to data cells, rather than directly applying styles to cells. It is, however, probably still easier to use the Styler function api when you are not concerned about optimization.

This is sub-optimal:

```python
def2.style.apply(highlight_max, props='color:white;background-color:darkblue;', axis=0)\ .apply(highlight_max, props='color:white;background-color:pink;', axis=1)\ .apply(highlight_max, props='color:white;background-color:purple', axis=\None)
```

This is better:

```python
build = lambda x: pd.DataFrame(x, index=df2.index, columns=df2.columns)
cls1 = build(df2.apply(highlight_max, props='cls-1 ', axis=0))
cls2 = build(df2.apply(highlight_max, props='cls-2 ', axis=1, result_type='expand').\ ...values)
cls3 = build(highlight_max(df2, props='cls-3 '))
```
4. Don't use tooltips

Tooltips require cell_ids to work and they generate extra HTML elements for every data cell.

5. If every byte counts use string replacement

You can remove unnecessary HTML, or shorten the default class names by replacing the default css dict. You can read a little more about CSS below.

```python
[44]: my_css = {
    "row_heading": "",
    "col_heading": "",
    "index_name": "",
    "col": "c",
    "row": "r",
    "col_trim": "",
    "row_trim": "",
    "level": "1",
    "data": "",
    "blank": "",
}

html = Styler(df4, uuid_len=0, cell_ids=False)
html.set_table_styles([{
    'selector': 'td', 'props': props,
    {'selector': '.c1', 'props': 'color:green;'},
    {'selector': '.l0', 'props': 'color:blue;'}],
    css_class_names=my_css)

print(html.to_html())
```
2.16.10 Builtin Styles

Some styling functions are common enough that we’ve “built them in” to the Styler, so you don’t have to write them and apply them yourself. The current list of such functions is:

- `.highlight_null`: for use with identifying missing data.
- `.highlight_min` and `.highlight_max`: for use with identifying extremeties in data.
- `.highlight_between` and `.highlight_quantile`: for use with identifying classes within data.
- `.background_gradient`: a flexible method for highlighting cells based or their, or other, values on a numeric scale.
- `.text_gradient`: similar method for highlighting text based on their, or other, values on a numeric scale.
- `.bar`: to display mini-charts within cell backgrounds.

The individual documentation on each function often gives more examples of their arguments.

**Highlight Null**

```python
df2.iloc[0,2] = np.nan
df2.iloc[4,3] = np.nan
df2.loc[:4].style.highlight_null(null_color='yellow')
```

```python
<pandas.io.formats.style.Styler at 0x7f9e344a5610>
```

```python
<pandas.io.formats.style.Styler at 0x7f9e344a54f0>
```

Highlight Min or Max

```python
[47]: df2.loc[:4].style.highlight_max(axis=1, props='color:white; font-weight:bold; background-color:darkblue;')
[47]: <pandas.io.formats.style.Styler at 0x7f9e34542370>
```

Highlight Between

This method accepts ranges as float, or NumPy arrays or Series provided the indexes match.

```python
[48]: left = pd.Series([1.0, 0.0, 1.0], index=['A', 'B', 'D'])
df2.loc[:4].style.highlight_between(left=left, right=1.5, axis=1, props='color:white; background-color:purple;')
[48]: <pandas.io.formats.style.Styler at 0x7f9e344afb80>
```

Highlight Quantile

Useful for detecting the highest or lowest percentile values

```python
[49]: df2.loc[:4].style.highlight_quantile(q_left=0.85, axis=None, color='yellow')
[49]: <pandas.io.formats.style.Styler at 0x7f9e344bcdf0>
```

Background Gradient and Text Gradient

You can create “heatmaps” with the `background_gradient` and `text_gradient` methods. These require matplotlib, and we’ll use Seaborn to get a nice colormap.

```python
[50]: import seaborn as sns
cm = sns.light_palette("green", as_cmap=True)
df2.style.background_gradient(cmap=cm)
[50]: <pandas.io.formats.style.Styler at 0x7f9e344af520>

[51]: df2.style.text_gradient(cmap=cm)
[51]: <pandas.io.formats.style.Styler at 0x7f9e299cdd00>
```

`background_gradient` and `text_gradient` have a number of keyword arguments to customise the gradients and colors. See the documentation.
pandas: powerful Python data analysis toolkit, Release 1.4.2

Set properties
Use Styler.set_properties when the style doesn’t actually depend on the values. This is just a simple wrapper for
.applymap where the function returns the same properties for all cells.
[52]: df2.loc[:4].style.set_properties(**{'background-color': 'black',
'color': 'lawngreen',
'border-color': 'white'})
[52]: <pandas.io.formats.style.Styler at 0x7f9e299da820>

Bar charts
You can include “bar charts” in your DataFrame.
[53]: df2.style.bar(subset=['A', 'B'], color='#d65f5f')
[53]: <pandas.io.formats.style.Styler at 0x7f9e299da6d0>
Additional keyword arguments give more control on centering and positioning, and you can pass a list of
[color_negative, color_positive] to highlight lower and higher values or a matplotlib colormap.
To showcase an example here’s how you can change the above with the new align option, combined with setting vmin
and vmax limits, the width of the figure, and underlying css props of cells, leaving space to display the text and the
bars. We also use text_gradient to color the text the same as the bars using a matplotlib colormap (although in this
case the visualization is probably better without this additional effect).
[54]: df2.style.format('{:.3f}', na_rep="")\
.bar(align=0, vmin=-2.5, vmax=2.5, cmap="bwr", height=50,
width=60, props="width: 120px; border-right: 1px solid black;")\
.text_gradient(cmap="bwr", vmin=-2.5, vmax=2.5)
[54]: <pandas.io.formats.style.Styler at 0x7f9e29963670>
The following example aims to give a highlight of the behavior of the new align options:
[56]: HTML(head)
[56]: <IPython.core.display.HTML object>

2.16.11 Sharing styles
Say you have a lovely style built up for a DataFrame, and now you want to apply the same style to a second DataFrame.
Export the style with df1.style.export, and import it on the second DataFrame with df1.style.set
[57]: style1 = df2.style\
.applymap(style_negative, props='color:red;')\
.applymap(lambda v: 'opacity: 20%;' if (v < 0.3) and (v > -0.3) else None)\
.set_table_styles([{"selector": "th", "props": "color: blue;"}])\
.hide(axis="index")
style1
[57]: <pandas.io.formats.style.Styler at 0x7f9e2999d6a0>

764

Chapter 2. User Guide


Notice that you’re able to share the styles even though they’re data aware. The styles are re-evaluated on the new DataFrame they’ve been used upon.

### 2.16.12 Limitations

- DataFrame only (use `Series.to_frame().style`)
- The index and columns do not need to be unique, but certain styling functions can only work with unique indexes.
- No large repr, and construction performance isn’t great; although we have some **HTML optimizations**
- You can only apply styles, you can’t insert new HTML entities, except via subclassing.

### 2.16.13 Other Fun and Useful Stuff

Here are a few interesting examples.

**Widgets**

Styler interacts pretty well with widgets. If you’re viewing this online instead of running the notebook yourself, you’re missing out on interactively adjusting the color palette.

```python
from ipywidgets import widgets
@widgets.interact
def f(h_neg=(0, 359, 1), h_pos=(0, 359), s=(0., 99.9), l=(0., 99.9)):
    return df2.style.background_gradient(
        cmap=sns.palettes.diverging_palette(h_neg=h_neg, h_pos=h_pos, s=s, l=l, as_cmap=True)
    )
interactive(children=(IntSlider(value=179, description='h_neg', max=359),
    IntSlider(value=179, description='h_pos', max=359),
    IntSlider(value=179, description='h_l'),
    IntSlider(value=179, description='h_s')
)
```

**Magnify**

```python
def magnify():
    return [dict(selector="th",
                 props=[("font-size", "4pt")]),
            dict(selector="td",
                 props=[("padding", "0em 0em")]),
            dict(selector="th:hover",
                 props=[("font-size", "12pt")]),
            dict(selector="tr:hover td:hover",
                 props=[("max-width", '200px'),
                         ('font-size', '12pt')])
]
```
Sticky Headers

If you display a large matrix or DataFrame in a notebook, but you want to always see the column and row headers you can use the `.set_sticky` method which manipulates the table styles CSS.

```
bigdf = pd.DataFrame(np.random.randn(16, 100))
bigdf.style.set_sticky(axis="index")
```

It is also possible to stick MultiIndexes and even only specific levels.

```
bigdf.index = pd.MultiIndex.from_product([['A','B'],[0,1],[0,1,2,3]])
bigdf.style.set_sticky(axis="index", pixel_size=18, levels=[1,2])
```

HTML Escaping

Suppose you have to display HTML within HTML, that can be a bit of pain when the renderer can’t distinguish. You can use the escape formatting option to handle this, and even use it within a formatter that contains HTML itself.

```
df4 = pd.DataFrame([['<div></div>', '"other"', '<span></span>']])
df4.style
```

```
df4.style.format(escape="html")
```

```
df4.style.format("<a href="https://pandas.pydata.org" target="_blank">{}"></a>", escape="html")
```

766 Chapter 2. User Guide
2.16.14 Export to Excel

Some support (since version 0.20.0) is available for exporting styled DataFrames to Excel worksheets using the OpenPyXL or XlsxWriter engines. CSS2.2 properties handled include:

- background-color
- color
- font-family
- font-style
- font-weight
- text-align
- text-decoration
- vertical-align
- white-space: nowrap
- Currently broken: border-style, border-width, border-color and their {top, right, bottom, left variants}
- Only CSS2 named colors and hex colors of the form #rgb or #rrggbb are currently supported.
- The following pseudo CSS properties are also available to set excel specific style properties:
  - number-format

Table level styles, and data cell CSS-classes are not included in the export to Excel: individual cells must have their properties mapped by the Styler.apply and/or Styler.applymap methods.

```
[67]: df2.style.\n    applymap(style_negative, props='color:red;').\n    highlight_max(axis=0).\n    to_excel('styled.xlsx', engine='openpyxl')
```

A screenshot of the output:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1.329212</td>
<td>-0.31628</td>
<td>-0.99081</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-1.070816</td>
<td>-1.438713</td>
<td>0.564417</td>
<td>0.295722</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-1.626404</td>
<td>0.219565</td>
<td>0.678805</td>
<td>1.889273</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.961538</td>
<td>0.104011</td>
<td>-0.481165</td>
<td>0.850229</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1.453425</td>
<td>1.057737</td>
<td>0.165562</td>
<td>0.515018</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>-1.336936</td>
<td>0.562861</td>
<td>1.392855</td>
<td>-0.063328</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>0.121668</td>
<td>1.207603</td>
<td>-0.00204</td>
<td>1.627796</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>0.354493</td>
<td>1.037528</td>
<td>-0.385684</td>
<td>0.519818</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>1.686583</td>
<td>-1.325963</td>
<td>1.428984</td>
<td>-2.089354</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>-0.12982</td>
<td>0.631523</td>
<td>-0.586538</td>
<td>0.29072</td>
<td></td>
</tr>
</tbody>
</table>
2.16.15 Export to $\LaTeX$

There is support (since version 1.3.0) to export Styler to $\LaTeX$. The documentation for the `.to_latex` method gives further detail and numerous examples.

2.16.16 More About CSS and HTML

Cascading Style Sheet (CSS) language, which is designed to influence how a browser renders HTML elements, has its own peculiarities. It never reports errors: it just silently ignores them and doesn’t render your objects how you intend so can sometimes be frustrating. Here is a very brief primer on how Styler creates HTML and interacts with CSS, with advice on common pitfalls to avoid.

**CSS Classes and Ids**

The precise structure of the CSS class attached to each cell is as follows.

- Cells with Index and Column names include `index_name` and `level<k>` where `k` is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `level<k>` where `k` is the level in a MultiIndex
  - `row<m>` where `m` is the numeric position of the row
- Column label cells include
  - `col_heading`
  - `level<k>` where `k` is the level in a MultiIndex
  - `col<n>` where `n` is the numeric position of the column
- Data cells include
  - `data`
  - `row<m>`, where `m` is the numeric position of the cell.
  - `col<n>`, where `n` is the numeric position of the cell.
- Blank cells include `blank`
- Trimmed cells include `col_trim` or `row_trim`

The structure of the id is `T_uuid_level<k>_row<m>_col<n>` where `level<k>` is used only on headings, and headings will only have either `row<m>` or `col<n>` whichever is needed. By default we’ve also prepended each row/column identifier with a UUID unique to each DataFrame so that the style from one doesn’t collide with the styling from another within the same notebook or page. You can read more about the use of UUIDs in *Optimization*.

We can see example of the HTML by calling the `.to_html()` method.

```
[68]: print(pd.DataFrame([[1,2],[3,4]], index=['i1', 'i2'], columns=['c1', 'c2']).style.to_html())

```
CSS Hierarchies

The examples have shown that when CSS styles overlap, the one that comes last in the HTML render, takes precedence. So the following yield different results:

[69]: df4 = pd.DataFrame([['text']])
  df4.style.applymap(lambda x: 'color:green;')
  .applymap(lambda x: 'color:red;')

[69]: <pandas.io.formats.style.Styler at 0x7f9e289a1df0>

[70]: df4.style.applymap(lambda x: 'color:red;')
  .applymap(lambda x: 'color:green;')

[70]: <pandas.io.formats.style.Styler at 0x7f9e289a1ac0>

This is only true for CSS rules that are equivalent in hierarchy, or importance. You can read more about CSS specificity here but for our purposes it suffices to summarize the key points:

A CSS importance score for each HTML element is derived by starting at zero and adding:

- 1000 for an inline style attribute
- 100 for each ID
- 10 for each attribute, class or pseudo-class
- 1 for each element name or pseudo-element

Let’s use this to describe the action of the following configurations

[71]: df4.style.set_uuid('a_')
  .set_table_styles([{'selector': 'td', 'props': 'color:red;'}])
  .applymap(lambda x: 'color:green;')
This text is red because the generated selector \#T_a\_ td is worth 101 (ID plus element), whereas \#T_a_row0\_col0 is only worth 100 (ID), so is considered inferior even though in the HTML it comes after the previous.

In the above case the text is blue because the selector \#T_b\_ .cls-1 is worth 110 (ID plus class), which takes precedence.

Now we have created another table style this time the selector \#T_c\_ td.data (ID plus element plus class) gets bumped up to 111.

If your style fails to be applied, and its really frustrating, try the !important trump card.

Finally got that green text after all!

### 2.16.17 Extensibility

The core of pandas is, and will remain, its “high-performance, easy-to-use data structures”. With that in mind, we hope that DataFrame.style accomplishes two goals:

- Provide an API that is pleasing to use interactively and is “good enough” for many tasks
- Provide the foundations for dedicated libraries to build on

If you build a great library on top of this, let us know and we’ll link to it.
Subclassing

If the default template doesn’t quite suit your needs, you can subclass `Styler` and extend or override the template. We’ll show an example of extending the default template to insert a custom header before each table.

```python
from jinja2 import Environment, ChoiceLoader, FileSystemLoader
from IPython.display import HTML
from pandas.io.formats.style import Styler

We’ll use the following template:

```{.py}
with open("templates/myhtml.tpl") as f:
    print(f.read())

{% extends "html_table.tpl" %}
{% block table %}
<h1>{{ table_title|default("My Table") }}</h1>
{{ super() }}
{% endblock table %}
```

Now that we’ve created a template, we need to set up a subclass of `Styler` that knows about it.

```python
class MyStyler(Styler):
    env = Environment(
        loader=ChoiceLoader([
            FileSystemLoader("templates"),  # contains ours
            Styler.loader,  # the default
        ])
    )
    template_html_table = env.get_template("myhtml.tpl")
```

Notice that we include the original loader in our environment’s loader. That’s because we extend the original template, so the Jinja environment needs to be able to find it.

Now we can use that custom styler. It’s `__init__` takes a DataFrame.

```python
MyStyler(df3)
```

```
<__main__.MyStyler at 0x7f9e28960760>
```

Our custom template accepts a `table_title` keyword. We can provide the value in the `.to_html` method.

```python
HTML(MyStyler(df3).to_html(table_title="Extending Example"))

```

```
<IPython.core.display.HTML object>
```

For convenience, we provide the `Styler.from_custom_template` method that does the same as the custom subclass.

```python
EasyStyler = Styler.from_custom_template("templates", "myhtml.tpl")
HTML(EasyStyler(df3).to_html(table_title="Another Title"))
```

```
<IPython.core.display.HTML object>
```
Template Structure

Here’s the template structure for the both the style generation template and the table generation template:

Style template:

| [82]: HTML(style_structure) |
| [82]: <IPython.core.display.HTML object> |

Table template:

| [84]: HTML(table_structure) |
| [84]: <IPython.core.display.HTML object> |

See the template in the GitHub repo for more details.

2.17 Computational tools

2.17.1 Statistical functions

Percent change

Series and DataFrame have a method `pct_change()` to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values before computing the percent change).

```
In [1]: ser = pd.Series(np.random.randn(8))
In [2]: ser.pct_change()
Out[2]:
   0     NaN
   1   -1.602976
   2    4.334938
   3   -0.247456
   4  -2.067345
   5  -1.142903
   6  -1.688214
   7  -9.759729
   dtype: float64
```

```
In [4]: df.pct_change(periods=3)
Out[4]:
            0     1     2     3
   0    NaN   NaN   NaN   NaN
   1    NaN   NaN   NaN   NaN
   2    NaN   NaN   NaN   NaN
   3   -0.218320 -1.054001 1.987147 -0.510183
   4  -0.439121 -1.816454 0.649715 -4.822809
   5  -0.127833 -3.042065 -5.866604 -1.776977
```

(continues on next page)
Covariance

*Series.cov()* can be used to compute covariance between series (excluding missing values).

```python
In [5]: s1 = pd.Series(np.random.randn(1000))
In [6]: s2 = pd.Series(np.random.randn(1000))
In [7]: s1.cov(s2)
Out[7]: 0.0006801088174310875
```

Analogously, *DataFrame.cov()* to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

**Note:** Assuming the missing data are missing at random this results in an estimate for the covariance matrix which is unbiased. However, for many applications this estimate may not be acceptable because the estimated covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimated correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See *Estimation of covariance matrices* for more details.

```python
In [8]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [9]: frame.cov()
Out[9]:
   a       b       c       d       e
a 1.000882 -0.003177 -0.002698 -0.006889 0.031912
b -0.003177 1.024721 0.000191 0.009212 0.000857
c -0.002698 0.000191 0.950735 -0.031743 -0.005087
d -0.006889 0.009212 -0.031743 1.002983 -0.047952
e 0.031912 0.000857 -0.005087 -0.047952 1.042487
```

*DataFrame.cov* also supports an optional *min_periods* keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```python
In [10]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])
In [11]: frame.loc[frame.index[:5], 'a'] = np.nan
In [12]: frame.loc[frame.index[5:10], 'b'] = np.nan
In [13]: frame.cov()
Out[13]:
   a       b       c
a 1.123670 -0.412851 0.018169
b -0.412851 1.154141 0.305260
c 0.018169 0.305260 1.301149
```
Correlation

Correlation may be computed using the `corr()` method. Using the `method` parameter, several methods for computing correlations are provided:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pearson</td>
<td>Standard correlation coefficient</td>
</tr>
<tr>
<td>kendall</td>
<td>Kendall Tau correlation coefficient</td>
</tr>
<tr>
<td>spearman</td>
<td>Spearman rank correlation coefficient</td>
</tr>
</tbody>
</table>

All of these are currently computed using pairwise complete observations. Wikipedia has articles covering the above correlation coefficients:

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearman’s rank correlation coefficient

**Note:** Please see the caveats associated with this method of calculating correlation matrices in the **covariance section**.

```python
In [15]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=["a", "b", "c", "d", "e"])
In [16]: frame.iloc[::2] = np.nan

# Series with Series
In [17]: frame["a"].corr(frame["b"])
Out[17]: 0.013479040400098775

In [18]: frame["a"].corr(frame["b"], method="spearman")
Out[18]: -0.007289885159540637

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:
    a  b     c  d     e
a 1.000000 0.013479 -0.049269 -0.042239 -0.028525
b 0.013479 1.000000 -0.020433 -0.011139 0.005654
c -0.049269 -0.020433 1.000000 0.018587 -0.054269
d -0.042239 -0.011139 0.018587 1.000000 -0.017060
e -0.028525 0.005654 -0.054269 -0.017060 1.000000
```
Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```python
In [20]: frame = pd.DataFrame(np.random.randn(20, 3), columns=["a", "b", "c"])
In [21]: frame.loc[frame.index[:5], "a"] = np.nan
In [22]: frame.loc[frame.index[5:10], "b"] = np.nan
In [23]: frame.corr()
Out[23]:
   a    b    c
a 1.0000 -0.1211 0.0695
b -0.1211 1.0000 0.0517
c  0.07  0.0517 1.0000
In [24]: frame.corr(min_periods=12)
Out[24]:
   a    b    c
a 1.0000 NaN 0.0695
b NaN 1.0000 0.0517
c  0.07  0.0517 1.0000
```

The `method` argument can also be a callable for a generic correlation calculation. In this case, it should be a single function that produces a single value from two ndarray inputs. Suppose we wanted to compute the correlation based on histogram intersection:

```python
# histogram intersection
In [25]: def histogram_intersection(a, b):
    .....:     return np.minimum(np.true_divide(a, a.sum()), np.true_divide(b, b.sum())).sum()
    .....:
In [26]: frame.corr(method=histogram_intersection)
Out[26]:
   a    b    c
a 1.0000 -6.4049 -2.0584
b -6.4049 1.0000 -19.2557
c -2.0584 -19.2557 1.0000
```

A related method `corrwith()` is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

```python
In [27]: index = ["a", "b", "c", "d", "e"]
In [28]: columns = ["one", "two", "three", "four"]
In [29]: df1 = pd.DataFrame(np.random.randn(5, 4), index=index, columns=columns)
In [30]: df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4], columns=columns)
In [31]: df1.corrwith(df2)
Out[31]:
```

(continues on next page)
one   -0.125501
two   -0.493244
three 0.344056
four  0.004183
dtype: float64

In [32]: df2.corrwith(df1, axis=1)
Out[32]:
a  -0.675817
b   0.458296
c   0.190809
d  -0.186275
e    NaN
dtype: float64

Data ranking

The \texttt{rank()} method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

\begin{verbatim}
In [33]: s = pd.Series(np.random.randn(5), index=list("abcde"))
In [34]: s["d"] = s["b"]  # so there's a tie
In [35]: s.rank()
Out[35]:
a  5.0
b  2.5
c  1.0
d  2.5
e  4.0
dtype: float64
\end{verbatim}

\texttt{rank()} is also a DataFrame method and can rank either the rows (\texttt{axis=0}) or the columns (\texttt{axis=1}). NaN values are excluded from the ranking.

\begin{verbatim}
In [36]: df = pd.DataFrame(np.random.randn(10, 6))
In [38]: df
Out[38]:
   0   1   2   3   4   5
0 -0.904948 -1.163537 -1.457187 0.135463 -1.457187 0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.038378 0.038378  1.140296
6  0.376892  0.959292  0.908114 -0.593740  0.038378  0.038378
7 -1.002601  1.957794 -0.120708  0.908114  0.038378  0.038378
8 -0.547231  0.664402 -0.519424 -0.73254  0.038378  0.038378
\end{verbatim}
rank optionally takes a parameter ascending which by default is true; when false, data is reverse-ranked, with larger values assigned a smaller rank.

rank supports different tie-breaking methods, specified with the method parameter:

- **average**: average rank of tied group
- **min**: lowest rank in the group
- **max**: highest rank in the group
- **first**: ranks assigned in the order they appear in the array

Windowing functions

See the window operations user guide for an overview of windowing functions.

### 2.18 Group by: split-apply-combine

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we might wish to do one of the following:

- **Aggregation**: compute a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means.
  - Compute group sizes / counts.
- **Transformation**: perform some group-specific computations and return a like-indexed object. Some examples:
  - Standardize data (zscore) within a group.
  - Filling NAs within groups with a value derived from each group.
• **Filtration**: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  
  – Discard data that belongs to groups with only a few members.
  
  – Filter out data based on the group sum or mean.

  • Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn’t fit into either of the above two categories.

Since the set of object instance methods on pandas data structures are generally rich and expressive, we often simply want to invoke, say, a DataFrame function on each group. The name GroupBy should be quite familiar to those who have used a SQL-based tool (or itertools), in which you can write code like:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

We aim to make operations like this natural and easy to express using pandas. We’ll address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the *cookbook* for some advanced strategies.

### 2.18.1 Splitting an object into groups

pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you may do the following:

```
In [1]: df = pd.DataFrame(
   ...:     [
   ...:         ("bird", "Falconiformes", 389.0),
   ...:         ("bird", "Psittaciformes", 24.0),
   ...:         ("mammal", "Carnivora", 80.2),
   ...:         ("mammal", "Primates", np.nan),
   ...:         ("mammal", "Carnivora", 58),
   ...:     ],
   ...:     index=["falcon", "parrot", "lion", "monkey", "leopard"],
   ...:     columns=("class", "order", "max_speed"),
   ...: )
```

```
In [2]: df
Out[2]:
    class  order      max_speed
   falcon  bird   Falconiformes     389.0
   parrot  bird   Psittaciformes     24.0
    lion   mammal    Carnivora       80.2
  monkey   mammal  Primates         NaN
  leopard  mammal    Carnivora       58.0
```

```
# default is axis=0
In [3]: grouped = df.groupby("class")
In [4]: grouped = df.groupby("order", axis="columns")
In [5]: grouped = df.groupby(["class", "order"])
```
The mapping can be specified many different ways:

- A Python function, to be called on each of the axis labels.
- A list or NumPy array of the same length as the selected axis.
- A dict or Series, providing a label -> group name mapping.
- For DataFrame objects, a string indicating either a column name or an index level name to be used to group.
- `df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`.
- A list of any of the above things.

Collectively we refer to the grouping objects as the keys. For example, consider the following DataFrame:

```
In [6]: df = pd.DataFrame(
...:     
...:     "A": ["foo", "bar", "foo", "bar", "foo", "foo", "foo"],
...:     "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
...:     "C": np.random.randn(8),
...:     "D": np.random.randn(8),
...:     }
...:     
...:     )
...:     
```

```
In [7]: df
Out[7]:
          A      B       C       D
0   foo   one  0.469112 -0.861849
1   bar   one -0.282863 -2.104569
2   foo   two -1.509059  0.494929
3   bar  three -1.135632  1.071804
4   foo   two  1.212112  0.721555
5   bar   two -0.173215 -0.706771
6   foo   one  0.119209 -1.039575
7   foo  three -1.044236  0.271860
```

On a DataFrame, we obtain a GroupBy object by calling `groupby()`. We could naturally group by either the A or B columns, or both:

```
In [8]: grouped = df.groupby("A")
In [9]: grouped = df.groupby(["A", "B"])
```

If we also have a MultiIndex on columns A and B, we can group by all but the specified columns

```
In [10]: df2 = df.set_index(["A", "B"])
In [11]: grouped = df2.groupby(level=df2.index.names.difference(["B"]))
In [12]: grouped.sum()
Out[12]:
```

(continues on next page)
These will split the DataFrame on its index (rows). We could also split by the columns:

```python
In [13]: def get_letter_type(letter):
    if letter.lower() in 'aeiou':
        return 'vowel'
    else:
        return 'consonant'

In [14]: grouped = df.groupby(get_letter_type, axis=1)
```

pandas `Index` objects support duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```python
In [15]: lst = [1, 2, 3, 1, 2, 3]
In [16]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)
In [17]: grouped = s.groupby(level=0)
In [18]: grouped.first()
Out[18]:
   1  2  3
  0  1  2  3
dtype: int64
In [19]: grouped.last()
Out[19]:
   1  10  20  30
  0  0  0  0
dtype: int64
In [20]: grouped.sum()
Out[20]:
   1  11  22  33
  0  3  5  6
dtype: int64
```

Note that no splitting occurs until it’s needed. Creating the GroupBy object only verifies that you’ve passed a valid mapping.

**Note:** Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though can’t be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.
GroupBy sorting

By default the group keys are sorted during the `groupby` operation. You may however pass `sort=False` for potential speedups:

```python
In [21]: df2 = pd.DataFrame({"X": ["B", "B", "A", "A"], "Y": [1, 2, 3, 4]})

In [22]: df2.groupby("X").sum()
Out[22]:
   Y
X  
A 7
B 3
```

```python
In [23]: df2.groupby("X", sort=False).sum()
Out[23]:
   Y
X  
B 3
A 7
```

Note that `groupby` will preserve the order in which observations are sorted within each group. For example, the groups created by `groupby()` below are in the order they appeared in the original DataFrame:

```python
In [24]: df3 = pd.DataFrame({"X": ["A", "B", "A", "B"], "Y": [1, 4, 3, 2]})

In [25]: df3.groupby("X").get_group("A")
Out[25]:
   X  Y
0  A  1
2  A  3

In [26]: df3.groupby("X").get_group("B")
Out[26]:
   X  Y
1  B  4
3  B  2
```

New in version 1.1.0.

GroupBy dropna

By default `NA` values are excluded from group keys during the `groupby` operation. However, in case you want to include `NA` values in group keys, you could pass `dropna=False` to achieve it.

```python
In [27]: df_list = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]

In [28]: df_dropna = pd.DataFrame(df_list, columns=["a", "b", "c"])

In [29]: df_dropna
Out[29]:
   a  b  c
0  1  2  0
```

(continues on next page)
The default setting of `dropna` argument is `True` which means `NA` are not included in group keys.

**GroupBy object attributes**

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```python
In [32]: df.groupby("A").groups
Out[32]: {"bar": [1, 3, 5], "foo": [0, 2, 4, 6, 7]}

In [33]: df.groupby(get_letter_type, axis=1).groups
Out[33]: {
    'consonant': ['B', 'C', 'D'],
    'vowel': ['A']
}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```python
In [34]: grouped = df.groupby(["A", "B"])

In [35]: grouped.groups
Out[35]: {
    ("bar", 'one'): [1],
    ("bar", 'three'): [3],
    ("bar", 'two'): [5],
    ("foo", 'one -- '): [0, 6],
    ("foo", 'three'): [7],
    ("foo", 'two'): [2, 4]
}

In [36]: len(grouped)
Out[36]: 6
```

GroupBy will tab complete column names (and other attributes):

```python
In [37]: df
Out[37]:
   height  weight gender
2000-01-01  42.849980  157.500553  male
```
GroupBy with MultiIndex

With **hierarchically-indexed data**, it's quite natural to group by one of the levels of the hierarchy.

Let's create a Series with a two-level MultiIndex.

```python
In [40]: arrays = [
    ....:     ['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
    ....:     ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'],
    ....:     ]
    ....:

In [41]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [42]: s = pd.Series(np.random.randn(8), index=index)

In [43]: s
Out[43]:
first  second
bar    one  -0.919854
       two  -0.042379
baz    one   1.247642
       two  -0.009920
foo    one   0.290213
       two   0.495767
qux    one   0.362949
       two   1.548106
dtype: float64
```
We can then group by one of the levels in `s`.

```python
In [44]: grouped = s.groupby(level=0)

In [45]: grouped.sum()
Out[45]:
first   bar  -0.962232
        baz  1.237723
        foo  0.785980
        qux  1.911055
       dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```python
In [46]: s.groupby(level="second").sum()
Out[46]:
second  one  0.980950
        two  1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```python
In [47]: s
Out[47]:
     first   second   third
    bar   doo   one  -1.131345
doo  two   -0.089329
    baz   bee   one   0.337863
doo  two   -0.945867
    foo   bop   one  -0.932132
doo  two   1.956030
    qux   bop   one   0.017587
doo  two  -0.016692
dtype: float64

In [48]: s.groupby(["first", "second"]).sum()
Out[48]:
      first   second
    bar   doo  -1.220674
    baz   bee  -0.608004
    foo   bop   1.023898
    qux   bop   0.000895
dtype: float64
```

Index level names may be supplied as keys.

```python
In [49]: s.groupby(["first", "second"]).sum()
Out[49]:
      first   second
    bar   doo  -1.220674
    baz   bee  -0.608004
    foo   bop   1.023898
    qux   bop   0.000895
```

(continues on next page)
More on the `sum` function and aggregation later.

### Grouping DataFrame with Index levels and columns

A DataFrame may be grouped by a combination of columns and index levels by specifying the column names as strings and the index levels as `pd.Grouper` objects.

```
In [50]: arrays = [
   ....:     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:     ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....:
   ....:
   ....:
   ....:

In [51]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [52]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3], 'B': np.arange(8)}, index=index)

In [53]: df
Out[53]:
   A B
first second
bar one 1 0
     two 1 1
baz one 1 2
     two 1 3
foo one 2 4
     two 2 5
qux one 3 6
     two 3 7
```

The following example groups `df` by the second index level and the `A` column.

```
In [54]: df.groupby([pd.Grouper(level=1), 'A']).sum()
Out[54]:
   B
second A
one 1 2
    2 4
    3 6
two 1 4
    2 5
    3 7
```

Index levels may also be specified by name.

```
In [55]: df.groupby([pd.Grouper(level="second"), 'A']).sum()
Out[55]:
   B
```

Index level names may be specified as keys directly to `groupby`.

```python
In [56]: df.groupby(["second", "A"]).sum()
Out[56]:
   second A
     B
one  1  2
    2  4
    3  6
two 1  4
    2  5
    3  7
```

### DataFrame column selection in `GroupBy`

Once you have created the `GroupBy` object from a DataFrame, you might want to do something different for each of the columns. Thus, using `[]` similar to getting a column from a DataFrame, you can do:

```python
In [57]: df = pd.DataFrame(
   ....:     {  
   ....:         "A": ["foo", "bar", "foo", "bar", "foo", "foo"],  
   ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],  
   ....:         "C": np.random.randn(8),  
   ....:         "D": np.random.randn(8),  
   ....:     }  
   ....: )
   ....: 
In [58]: grouped = df.groupby("
```

```python
In [59]: grouped_C = grouped["C"]
```

(continues on next page)
In [61]: grouped_D = grouped[“D”]

This is mainly syntactic sugar for the alternative and much more verbose:

In [62]: df[“C”].groupby(df[“A”])
Out[62]:<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f2d8eb90a00>

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

### 2.18.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

In [63]: grouped = df.groupby(‘A’)

In [64]: for name, group in grouped:
   ....:    print(name)
   ....:    print(group)
   ....:
   bar
   A  B  C  D
   1 bar one 0.254161 1.511763
   3 bar three 0.215897 -0.990582
   5 bar two -0.077118 1.211526
   foo
   A  B  C  D
   0 foo one -0.575247 1.346061
   2 foo two -1.143704 1.627081
   4 foo two 1.193555 -0.441652
   6 foo one -0.408530 0.268520
   7 foo three -0.862495 0.024580

In the case of grouping by multiple keys, the group name will be a tuple:

In [65]: for name, group in df.groupby([‘A’, ‘B’]):
   ....:    print(name)
   ....:    print(group)
   ....:
   (‘bar’, ‘one’)
   A  B  C  D
   1 bar one 0.254161 1.511763
   (‘bar’, ‘three’)
   A  B  C  D
   3 bar three 0.215897 -0.990582
   (‘bar’, ‘two’)
   A  B  C  D
   5 bar two -0.077118 1.211526
   (‘foo’, ‘one’)
   A  B  C  D
   0 foo one -0.575247 1.346061

(continues on next page)
2.18.3 Selecting a group

A single group can be selected using `get_group()`:

```python
In [66]: grouped.get_group("bar")
Out[66]:
    A   B   C   D
1 bar one 0.254161 1.511763
3 bar three 0.215897 -0.990582
5 bar two -0.077118 1.211526
```

Or for an object grouped on multiple columns:

```python
In [67]: df.groupby(["A", "B"])
Out[67]:
    A   B   C   D
1 bar one 0.254161 1.511763
```

2.18.4 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. These operations are similar to the aggregating API, window API, and resample API.

An obvious one is aggregation via the `aggregate()` or equivalently `agg()` method:

```python
In [68]: grouped = df.groupby("A")
In [69]: grouped.aggregate(np.sum)
Out[69]:
    C   D
A
bar  0.392940 1.732707
foo -1.796421 2.824590

In [70]: grouped = df.groupby(["A", "B"])
In [71]: grouped.aggregate(np.sum)
Out[71]:
    C   D
A
```

See `Iterating through groups`.
As you can see, the result of the aggregation will have the group names as the new index along the grouped axis. In the case of multiple keys, the result is a \textit{MultiIndex} by default, though this can be changed by using the \texttt{as_index} option:

\begin{verbatim}
In [72]: grouped = df.groupby(["A", "B"], as_index=False)
In [73]: grouped.aggregate(np.sum)
Out[73]:
        A   B   C    D
   0  bar  one  0.254161  1.511763
   1  bar  three  0.215897  0.990582
   2  bar  two  -0.077118  1.211526
   3  foo  one  -0.983776  1.614581
   4  foo  three  -0.862495   0.024580
   5  foo  two   0.049851  1.185429
\end{verbatim}

\begin{verbatim}
In [74]: df.groupby("A", as_index=False).sum()
Out[74]:
       A   C    D
   0  bar  0.392940  1.732707
   1  foo -1.796421  2.824590
\end{verbatim}

Note that you could use the \texttt{reset_index} DataFrame function to achieve the same result as the column names are stored in the resulting \textit{MultiIndex}:

\begin{verbatim}
In [75]: df.groupby(["A", "B"] ).sum().reset_index()
Out[75]:
        A   B   C    D
   0  bar  one  0.254161  1.511763
   1  bar  three  0.215897  0.990582
   2  bar  two  -0.077118  1.211526
   3  foo  one  -0.983776  1.614581
   4  foo  three  -0.862495   0.024580
   5  foo  two   0.049851  1.185429
\end{verbatim}

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the \texttt{size} method. It returns a Series whose index are the group names and whose values are the sizes of each group.

\begin{verbatim}
In [76]: grouped.size()
Out[76]:
        A    B  size
   0  bar  one    1
   1  bar  three    1
   2  bar  two    1
   3  foo  one    2
   4  foo  three    1
   5  foo  two    2
\end{verbatim}
```python
In [77]: grouped.describe()
Out[77]:
      C    ...    D
     ...    ...
     count  mean  std  min  25%  50%  ...  std  min
-25%  511763  1.511763 1.511763 0.254161 0.254161 0.254161 1.511763 1.511763
  1    1.0  0.215897 NaN  0.215897 0.215897 0.215897 NaN -0.990582 -0.990582
  2    1.0 -0.077118 NaN -0.077118 -0.077118 -0.077118 NaN 1.211526 1.211526
  3    2.0 -0.491888 0.117887 -0.575247 -0.533567 -0.491888 0.761937 0.268520 0.
  4    1.0 -0.862495 NaN -0.862495 -0.862495 -0.862495 NaN 0.024580 0.
  5    2.0  0.024925 1.652692 -1.143704 -0.559389 0.024925 1.462816 -0.441652 0.

[6 rows x 16 columns]
```

Another aggregation example is to compute the number of unique values of each group. This is similar to the `value_counts` function, except that it only counts unique values.

```python
In [78]: ll = [['foo', 1], ['foo', 2], ['foo', 2], ['bar', 1], ['bar', 1]]
In [79]: df4 = pd.DataFrame(ll, columns=['A', 'B'])
In [80]: df4
Out[80]:
   A  B
0 foo 1
1 foo 2
2 foo 2
3 bar 1
4 bar 1
```

```python
In [81]: df4.groupby("A")['B'].nunique()
Out[81]:
Name: B, dtype: int64
```

**Note:** Aggregation functions will not return the groups that you are aggregating over if they are named `columns`, when `as_index=True`, the default. The grouped columns will be the `indices` of the returned object.

Passing `as_index=False` will return the groups that you are aggregating over, if they are named `columns`.

Aggregating functions are the ones that reduce the dimension of the returned objects. Some common aggregating functions are tabulated below:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean()</td>
<td>Compute mean of groups</td>
</tr>
<tr>
<td>sum()</td>
<td>Compute sum of group values</td>
</tr>
<tr>
<td>size()</td>
<td>Compute group sizes</td>
</tr>
<tr>
<td>count()</td>
<td>Compute count of group</td>
</tr>
<tr>
<td>std()</td>
<td>Standard deviation of groups</td>
</tr>
<tr>
<td>var()</td>
<td>Compute variance of groups</td>
</tr>
<tr>
<td>sem()</td>
<td>Standard error of the mean of groups</td>
</tr>
<tr>
<td>describe()</td>
<td>Generates descriptive statistics</td>
</tr>
<tr>
<td>first()</td>
<td>Compute first of group values</td>
</tr>
<tr>
<td>last()</td>
<td>Compute last of group values</td>
</tr>
<tr>
<td>nth()</td>
<td>Take nth value, or a subset if n is a list</td>
</tr>
<tr>
<td>min()</td>
<td>Compute min of group values</td>
</tr>
<tr>
<td>max()</td>
<td>Compute max of group values</td>
</tr>
</tbody>
</table>

The aggregating functions above will exclude NA values. Any function which reduces a `Series` to a scalar value is an aggregation function and will work, a trivial example is `df.groupby('A').agg(lambda ser: 1)`. Note that `nth()` can act as a reducer or a filter, see here.

### Applying multiple functions at once

With grouped `Series` you can also pass a list or dict of functions to do aggregation with, outputting a DataFrame:

```python
In [82]: grouped = df.groupby("A")
In [83]: grouped["C"].agg([np.sum, np.mean, np.std])
Out[83]:
   sum   mean   std
A  
bar 0.392940 0.130980 0.181231
foo -1.796421 -0.359284 0.912265
```

On a grouped `DataFrame`, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```python
In [84]: grouped["C", "D"].agg([np.sum, np.mean, np.std])
Out[84]:
   C   D
sum   mean   std   sum   mean   std
A  
bar  0.392940 0.130980 0.181231 1.732707 0.577569 1.366330
foo -1.796421 -0.359284 0.912265 2.824590 0.564918 0.884785
```

The resulting aggregations are named for the functions themselves. If you need to rename, then you can add in a chained operation for a `Series` like this:

```python
In [85]: (grouped["C"]
    ....: .agg([np.sum, np.mean, np.std])
    ....: .rename(columns={"sum": "foo", "mean": "bar", "std": "baz"})
    ....: )
```

(continues on next page)
For a grouped DataFrame, you can rename in a similar manner:

```python
In [86]: (grouped[['C', 'D']].agg([np.sum, np.mean, np.std]).rename(  
            columns={"sum": "foo", "mean": "bar", "std": "baz"})
  )
```

```
Out[86]:
         C   D
    foo  bar  baz  foo  bar  baz
A  
  bar   0.39294  0.13098  0.18123  1.73270  0.57757  1.36633
  foo  -1.79642 -0.35928  0.91226  2.82459  0.56492  0.88479
```

**Note:** In general, the output column names should be unique. You can’t apply the same function (or two functions with the same name) to the same column.

```python
In [87]: grouped["C"].agg(["sum", "sum"])
```

```
Out[87]:
          sum  sum
A  
  bar    0.39294  0.39294
  foo  -1.79642 -1.79642
```

Pandas does allow you to provide multiple lambdas. In this case, pandas will mangle the name of the (nameless) lambda functions, appending `<i>` to each subsequent lambda.

```python
In [88]: grouped["C"].agg([lambda x: x.max() - x.min(), lambda x: x.median() - x.mean()])
```

```
Out[88]:
      <lambda_0>  <lambda_1>
A  
  bar   0.33128   0.08492
  foo   2.33726  -0.21596
```
**Named aggregation**

New in version 0.25.0.

To support column-specific aggregation with control over the output column names, pandas accepts the special syntax in GroupBy. `agg()`, known as “named aggregation”, where

- The keywords are the output column names
- The values are tuples whose first element is the column to select and the second element is the aggregation to apply to that column. pandas provides the `pandas.NamedAgg` namedtuple with the fields ['column', 'aggfunc'] to make it clearer what the arguments are. As usual, the aggregation can be a callable or a string alias.

```
In [89]: animals = pd.DataFrame(
       ....:     {
       ....:         "kind": ["cat", "dog", "cat", "dog"],
       ....:         "height": [9.1, 6.0, 9.5, 34.0],
       ....:         "weight": [7.9, 7.5, 9.9, 198.0],
       ....:     }
       ....: )

In [90]: animals
Out[90]:
    kind    height    weight
0    cat     9.1      7.9
1   dog      6.0      7.5
2    cat     9.5      9.9
3   dog    34.0    198.0

In [91]: animals.groupby("kind").agg(
       ....:     min_height=pd.NamedAgg(column="height", aggfunc="min"),
       ....:     max_height=pd.NamedAgg(column="height", aggfunc="max"),
       ....:     average_weight=pd.NamedAgg(column="weight", aggfunc=np.mean),
       ....: )

Out[91]:
    min_height  max_height  average_weight
kind
  cat     9.1        9.5      8.90
  dog     6.0      34.0    102.75
```

`pandas.NamedAgg` is just a namedtuple. Plain tuples are allowed as well.

```
In [92]: animals.groupby("kind").agg(
       ....:     min_height=("height", "min"),
       ....:     max_height=("height", "max"),
       ....:     average_weight=("weight", np.mean),
       ....: )

Out[92]:
    min_height  max_height  average_weight
kind
  cat     9.1        9.5      8.90
  dog     6.0      34.0    102.75
```

2.18. Group by: split-apply-combine
If your desired output column names are not valid Python keywords, construct a dictionary and unpack the keyword arguments:

```python
animals.groupby("kind").agg(
    **{
        "total weight": pd.NamedAgg(column="weight", aggfunc=sum)
    }
)
```

```plaintext
Out[93]:
    kind  total weight
  cat     17.8
  dog    205.5
```

Additional keyword arguments are not passed through to the aggregation functions. Only pairs of (column, aggfunc) should be passed as **kwargs. If your aggregation functions requires additional arguments, partially apply them with `functools.partial()`.

**Note:** For Python 3.5 and earlier, the order of **kwargs in a function was not preserved. This means that the output column ordering would not be consistent. To ensure consistent ordering, the keys (and so output columns) will always be sorted for Python 3.5.

Named aggregation is also valid for Series groupby aggregations. In this case there's no column selection, so the values are just the functions:

```python
animals.groupby("kind").height.agg(
    **{
        "min_height": "min",
        "max_height": "max",
    }
)
```

```plaintext
Out[94]:
    kind  min_height  max_height
  cat      9.1       9.5
  dog      6.0      34.0
```

### Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```python
grouped.agg({"C": np.sum, "D": lambda x: np.std(x, ddof=1)})
```

```plaintext
Out[95]:
    C      D
  A  0.392940  1.366330
  B -1.796421  0.884785
```

The function names can also be strings. In order for a string to be valid it must be either implemented on GroupBy or available via dispatching:
In [96]: grouped.agg({"C": "sum", "D": "std"})
Out[96]:
     C     D
A
bar  0.392940  1.366330
foo -1.796421  0.884785

Cython-optimized aggregation functions

Some common aggregations, currently only sum, mean, std, and sem, have optimized Cython implementations:

In [97]: df.groupby("A").sum()
Out[97]:
     C     D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [98]: df.groupby(["A", "B"]).mean()
Out[98]:
     C     D
A B
bar one  0.254161  1.511763
two -0.077118  1.211526
three  0.215897  -0.990582
foo one -0.491888  0.807291
two 0.024925  0.592714
three -0.862495  0.024580

Of course sum and mean are implemented on pandas objects, so the above code would work even without the special versions via dispatching (see below).

Aggregations with User-Defined Functions

Users can also provide their own functions for custom aggregations. When aggregating with a User-Defined Function (UDF), the UDF should not mutate the provided Series, see Mutating with User Defined Function (UDF) methods for more information.

In [99]: animals.groupby("kind")[["height"]].agg(lambda x: set(x))
Out[99]:
   height
kind
cat  {9.1, 9.5}
dog  {34.0, 6.0}

The resulting dtype will reflect that of the aggregating function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as DataFrame construction.

In [100]: animals.groupby("kind")[["height"]].agg(lambda x: x.astype(int).sum())
Out[100]:
   height
(continues on next page)
kind
**cat** 18
**dog** 40

### 2.18.5 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. The transform function must:

- Return a result that is either the same size as the group chunk or broadcastable to the size of the group chunk (e.g., a scalar, `grouped.transform(lambda x: x.iloc[-1])`).
- Operate column-by-column on the group chunk. The transform is applied to the first group chunk using `chunk.apply`.
- Not perform in-place operations on the group chunk. Group chunks should be treated as immutable, and changes to a group chunk may produce unexpected results. For example, when using `fillna`, `inplace` must be `False` (`grouped.transform(lambda x: x.fillna(inplace=False))`).
- (Optionally) operates on the entire group chunk. If this is supported, a fast path is used starting from the second chunk.

Similar to *Aggregations with User-Defined Functions*, the resulting dtype will reflect that of the transformation function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as `DataFrame` construction.

Suppose we wished to standardize the data within each group:

```python
In [101]: index = pd.date_range("10/1/1999", periods=1100)
In [102]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
In [103]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()
In [104]: ts.head()
Out[104]:
2000-01-08  0.779333
2000-01-09  0.778852
2000-01-10  0.786476
2000-01-11  0.782797
2000-01-12  0.798110
Freq: D, dtype: float64
In [105]: ts.tail()
Out[105]:
2002-09-30  0.660294
2002-10-01  0.631095
2002-10-02  0.673601
2002-10-03  0.709213
2002-10-04  0.719369
Freq: D, dtype: float64
In [106]: transformed = ts.groupby(lambda x: x.year).transform(
....:
... lambda x: (x - x.mean()) / x.std()
```
(continues on next page)
We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```python
# Original Data
In [107]: grouped = ts.groupby(lambda x: x.year)

In [108]: grouped.mean()
Out[108]:
   2000   0.442441
   2001   0.526246
   2002   0.459365
dtype: float64

In [109]: grouped.std()
Out[109]:
   2000   0.131752
   2001   0.210945
   2002   0.128753
dtype: float64

# Transformed Data
In [110]: grouped_trans = transformed.groupby(lambda x: x.year)

In [111]: grouped_trans.mean()
Out[111]:
   2000  1.193722e-15
   2001  1.945476e-15
   2002  1.272949e-15
dtype: float64

In [112]: grouped_trans.std()
Out[112]:
   2000   1.0
   2001   1.0
   2002   1.0
dtype: float64
```

We can also visually compare the original and transformed data sets.

```python
In [113]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})

In [114]: compare.plot()
Out[114]: <AxesSubplot:>
```
Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```
In [115]: ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
Out[115]:
2000-01-08  0.623893
2000-01-09  0.623893
2000-01-10  0.623893
2000-01-11  0.623893
2000-01-12  0.623893
...   
2002-09-30  0.558275
2002-10-01  0.558275
2002-10-02  0.558275
2002-10-03  0.558275
2002-10-04  0.558275
Freq: D, Length: 1001, dtype: float64
```

Alternatively, the built-in methods could be used to produce the same outputs.

```
In [116]: max = ts.groupby(lambda x: x.year).transform("max")
In [117]: min = ts.groupby(lambda x: x.year).transform("min")
In [118]: max - min
```
Another common data transform is to replace missing data with the group mean.

```
In [119]: data_df
Out[119]:
   A       B       C
0  1.539708 -1.166480  0.533026
1  1.302092 -0.505754   NaN
2 -1.309622  1.104803 -1.161657
3 -1.924296  0.396437  0.812436
4  0.234564  0.517098  0.393534
...     ...     ...
995 -0.093110  0.683847 -0.774753
996 -0.185043  1.438572   NaN
997 -0.394469 -0.642343  0.011374
998 -1.174126  1.857148   NaN
999  0.234564  0.517098  0.393534
[1000 rows x 3 columns]
```

```
In [120]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [121]: key = countries[np.random.randint(0, 4, 1000)]

In [122]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [123]: grouped.count()
Out[123]:
     A    B    C
  GR  209  217  189
  JP  240  255  217
  UK  216  231  193
  US  239  250  217

In [124]: transformed = grouped.transform(lambda x: x.fillna(x.mean()))
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.
In [125]: grouped_trans = transformed.groupby(key)

In [126]: grouped.mean()  # original group means
Out[126]:
   A  B  C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [127]: grouped_trans.mean()  # transformation did not change group means
Out[127]:
   A  B  C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [128]: grouped.count()  # original has some missing data points
Out[128]:
   A  B  C
GR 209 217 189
JP 240 255 217
UK 216 231 193
US 239 250 217

In [129]: grouped_trans.count()  # counts after transformation
Out[129]:
   A  B  C
GR 228 228 228
JP 267 267 267
UK 247 247 247
US 258 258 258

In [130]: grouped_trans.size()  # Verify non-NA count equals group size
Out[130]:
      GR  JP  UK  US
size: 228  267  247  258

dtype: int64

Note: Some functions will automatically transform the input when applied to a GroupBy object, but returning an object of the same shape as the original. Passing as_index=False will not affect these transformation methods.

For example: fillna, ffill, bfill, shift...

In [131]: grouped.ffill()  
Out[131]:
   A   B   C
0  1.539708 -1.166480  0.533026
1  1.302092 -0.505754  0.533026

(continues on next page)
Window and resample operations

It is possible to use `resample()`, `expanding()` and `rolling()` as methods on groupbys.

The example below will apply the `rolling()` method on the samples of the column `B` based on the groups of column `A`.

```python

In [133]: df_re.groupby("A").rolling(4).B.mean()
```

```python
Out[133]:
```

(continues on next page)
The expanding() method will accumulate a given operation (sum() in the example) for all the members of each particular group.

In [135]: df_re.groupby("A").expanding().sum()
Out[135]:

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>1.0</td>
<td>3.0</td>
<td>6.0</td>
<td>10.0</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>15.0</td>
<td>91.0</td>
<td>108.0</td>
<td>126.0</td>
<td>145.0</td>
<td>...</td>
</tr>
</tbody>
</table>

[20 rows x 1 columns]

Suppose you want to use the resample() method to get a daily frequency in each group of your dataframe and wish to complete the missing values with the ffill() method.

In [136]: df_re = pd.DataFrame(
    .....: {  
    .....:     "date": pd.date_range(start="2016-01-01", periods=4, freq="W"),  
    .....:     "group": [1, 1, 2, 2],  
    .....:     "val": [5, 6, 7, 8],  
    .....: }  
    .....: ).set_index("date")  
    .....: )

In [137]: df_re
Out[137]:

<table>
<thead>
<tr>
<th>group</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
</tr>
<tr>
<td>2016-01-03</td>
<td>1 5</td>
</tr>
<tr>
<td>2016-01-10</td>
<td>1 6</td>
</tr>
<tr>
<td>2016-01-17</td>
<td>2 7</td>
</tr>
<tr>
<td>2016-01-24</td>
<td>2 8</td>
</tr>
</tbody>
</table>

In [138]: df_re.groupby("group").resample("1D").ffill()
Out[138]:

<table>
<thead>
<tr>
<th>group</th>
<th>date</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2016-01-03</td>
<td>1 5</td>
</tr>
<tr>
<td>1</td>
<td>2016-01-04</td>
<td>1 5</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

2016-01-05
2016-01-06
2016-01-07
...
2

2016-01-20
2016-01-21
2016-01-22
2016-01-23
2016-01-24

1
1
1
...
2
2
2
2
2

5
5
5
...
7
7
7
7
8

[16 rows x 2 columns]

2.18.6 Filtration
The filter method returns a subset of the original object. Suppose we want to take only elements that belong to
groups with a group sum greater than 2.
In [139]: sf = pd.Series([1, 1, 2, 3, 3, 3])
In [140]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[140]:
3
3
4
3
5
3
dtype: int64
The argument of filter must be a function that, applied to the group as a whole, returns True or False.
Another useful operation is filtering out elements that belong to groups with only a couple members.
In [141]: dff = pd.DataFrame({"A": np.arange(8), "B": list("aabbbbcc")})
In [142]: dff.groupby("B").filter(lambda x: len(x) > 2)
Out[142]:
A B
2 2 b
3 3 b
4 4 b
5 5 b
Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do
not pass the filter are filled with NaNs.
In [143]: dff.groupby("B").filter(lambda x: len(x) > 2, dropna=False)
Out[143]:
A
B
0 NaN NaN
1 NaN NaN
2 2.0
b
3 3.0
b
4 4.0
b
5 5.0
b
(continues on next page)

2.18. Group by: split-apply-combine

803


For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

```python
In [144]: dff["C"] = np.arange(8)
In [145]: dff.groupby("B").filter(lambda x: len(x["C"]) > 2)
Out[145]:
A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
```

**Note:** Some functions when applied to a groupby object will act as a filter on the input, returning a reduced shape of the original (and potentially eliminating groups), but with the index unchanged. Passing `as_index=False` will not affect these transformation methods.

For example: head, tail.

```python
In [146]: dff.groupby("B").head(2)
Out[146]:
A  B  C
0  0  a  0
1  1  a  1
2  2  b  2
3  3  b  3
6  6  c  6
7  7  c  7
```

### 2.18.7 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```python
In [147]: grouped = df.groupby("A")

In [148]: grouped.agg(lambda x: x.std())
Out[148]:
     C      D
A bar  0.181231  1.366330
  foo  0.912265  0.884785
```

But, it’s rather verbose and can be untidy if you need to pass additional arguments. Using a bit of metaprogramming cleverness, GroupBy now has the ability to “dispatch” method calls to the groups:

```python
In [149]: grouped.std()
Out[149]:
        C       D
bar  0.181231  1.366330
foo  0.912265  0.884785
```
What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```python
In [150]: tsdf = pd.DataFrame(
    .....:     np.random.randn(1000, 3),
    .....:     index=pd.date_range("1/1/2000", periods=1000),
    .....:     columns=["A", "B", "C"],
    .....: )

In [151]: tsdf.iloc[::2] = np.nan

In [152]: grouped = tsdf.groupby(lambda x: x.year)

In [153]: grouped.fillna(method="pad")
```

```
Out[153]:
    A         B         C
2000-01-01 NaN     NaN       NaN
2000-01-02 -0.353501 -0.080957 -0.876864
2000-01-03 -0.353501 -0.080957 -0.876864
2000-01-04  0.050976  0.044273 -0.559849
2000-01-05  0.050976  0.044273 -0.559849
2002-09-22  0.005011  0.053897 -1.026922
2002-09-23  0.005011  0.053897 -1.026922
2002-09-24 -0.456542 -1.849051  1.559856
2002-09-25 -0.456542 -1.849051  1.559856
2002-09-26  1.123162  0.354660  1.128135
[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called `fillna` on the groups.

The `nlargest` and `nsmallest` methods work on Series style groupbys:

```python
In [154]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])

In [155]: g = pd.Series(list("abababab"))

In [156]: gb = s.groupby(g)

In [157]: gb.nlargest(3)
```

```
Out[157]:
      a
a  4    19.0
  0  9.0
```

(continues on next page)
2.18.8 Flexible apply

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want GroupBy to infer how to combine the results. For these, use the apply function, which can be substituted for both aggregate and transform in many standard use cases. However, apply can handle some exceptional use cases, for example:

```python
In [159]: df
Out[159]:
     A   B     C     D
0   foo one -0.575247  1.346061
1   bar one  0.254161  1.511763
2   foo two -1.143704  1.627081
3   bar three  0.215897 -0.990582
4   foo two  1.193555 -0.441652
5   bar two  0.077118  1.211526
6   foo one  0.408530  0.268520
7   foo three  0.862495  0.024580

In [160]: grouped = df.groupby("A")
# could also just call .describe()
In [161]: grouped["C"].apply(lambda x: x.describe())
Out[161]:
       A
bar     count 3.000000
         mean  0.130980
         std  0.181231
         min -0.077118
         25%  0.069390
         ...
foo     min -1.143704
         25%  0.862495
         50%  0.575247
         75%  0.408530
```
(continues on next page)
The dimension of the returned result can also change:

```python
In [162]: grouped = df.groupby('A')['C']
```

```python
In [163]: def f(group):
   ....:     return pd.DataFrame({'original': group,
   ....:                              'demeaned': group - group.mean()})
   ....:
```

```python
In [164]: grouped.apply(f)
Out[164]:
          original  demeaned
0  -0.575247  -0.215962
1   0.254161   0.123181
2 -1.143704  -0.784420
3   0.215897   0.084917
4  1.193555   1.552839
5  -0.077118  -0.208098
6 -0.408530  -0.049245
7 -0.862495  -0.503211
```

apply on a Series can operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame:

```python
In [165]: def f(x):
   ....:     return pd.Series([x, x ** 2], index=['x', 'x^2'])
   ....:
```

```python
In [166]: s = pd.Series(np.random.rand(5))
```

```python
In [167]: s
Out[167]:
0  0.321438
1  0.493496
2  0.139505
3  0.910103
4  0.194158
dtype: float64
```

```python
In [168]: s.apply(f)
Out[168]:
          x    x^2
0  0.321438  0.103323
1  0.493496  0.243538
2  0.139505  0.019462
3  0.910103  0.828287
4  0.194158  0.037697
```

Note: apply can act as a reducer, transformer, or filter function, depending on exactly what is passed to it. So
depending on the path taken, and exactly what you are grouping. Thus the grouped columns(s) may be included in the output as well as set the indices.

Similar to Aggregations with User-Defined Functions, the resulting dtype will reflect that of the apply function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as DataFrame construction.

### 2.18.9 Numba Accelerated Routines

New in version 1.1.

If Numba is installed as an optional dependency, the transform and aggregate methods support engine='numba' and engine_kwags arguments. See enhancing performance with Numba for general usage of the arguments and performance considerations.

The function signature must start with values, index exactly as the data belonging to each group will be passed into values, and the group index will be passed into index.

**Warning:** When using engine='numba', there will be no “fall back” behavior internally. The group data and group index will be passed as NumPy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

### 2.18.10 Other useful features

**Automatic exclusion of “nuisance” columns**

Again consider the example DataFrame we’ve been looking at:

```
In [169]: df
Out[169]:
   A     B   C     D
0  foo  one -0.575247  1.346061
1  bar  one   0.254161  1.511763
2  foo  two -1.143704  1.627081
3  bar  three   0.215897 -0.990582
4  foo  two   1.193555 -0.441652
5  bar  two  -0.077118  1.211526
6  foo  one  -0.408530  0.268520
7  foo  three -0.862495  0.024580
```

Suppose we wish to compute the standard deviation grouped by the A column. There is a slight problem, namely that we don’t care about the data in column B. We refer to this as a “nuisance” column. If the passed aggregation function can’t be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [170]: df.groupby("A").std()
Out[170]:
   C     D
A
bar   0.181231  1.366330
foo   0.912265  0.884785
```
Note that `df.groupby('A').colname.std().` is more efficient than `df.groupby('A').std().colname`, so if the result of an aggregation function is only interesting over one column (here `colname`), it may be filtered *before* applying the aggregation function.

**Note:** Any object column, also if it contains numerical values such as Decimal objects, is considered as a “nuisance” columns. They are excluded from aggregate functions automatically in groupby.

If you do wish to include decimal or object columns in an aggregation with other non-nuisance data types, you must do so explicitly.

```
In [171]: from decimal import Decimal

In [172]: df_dec = pd.DataFrame(
    .....:     {
    .....:       "id": [1, 2, 1, 2],
    .....:       "int_column": [1, 2, 3, 4],
    .....:       "dec_column": [
    .....:         Decimal("0.50"),
    .....:         Decimal("0.15"),
    .....:         Decimal("0.25"),
    .....:         Decimal("0.40"),
    .....:       ],
    .....:     }
    .....: )
    .....:

# Decimal columns can be sum'd explicitly by themselves...
In [173]: df_dec.groupby(["id"])[["dec_column"]].sum()
Out[173]:
    dec_column
   id
1   0.75
2   0.55

# ...but cannot be combined with standard data types or they will be excluded
In [174]: df_dec.groupby(["id"])[["int_column", "dec_column"]].sum()
Out[174]:
   int_column
   id
1   4
2   6

# Use .agg function to aggregate over standard and "nuisance" data types
# at the same time
In [175]: df_dec.groupby(["id"]).agg({"int_column": "sum", "dec_column": "sum"})
Out[175]:
   int_column  dec_column
   id
1   4          0.75
2   6          0.55
```
Handling of (un)observed Categorical values

When using a Categorical grouper (as a single grouper, or as part of multiple groupers), the observed keyword controls whether to return a cartesian product of all possible groupers values (observed=False) or only those that are observed groupers (observed=True).

Show all values:

```python
In [176]: pd.Series([1, 1, 1]).groupby(
    ....:     pd.Categorical(["a", "a", "a"], categories=["a", "b"]), observed=False
    ....: ).count()
    ....:
Out[176]:
a  3
b  0
dtype: int64
```

Show only the observed values:

```python
In [177]: pd.Series([1, 1, 1]).groupby(
    ....:     pd.Categorical(["a", "a", "a"], categories=["a", "b"]), observed=True
    ....: ).count()
    ....:
Out[177]:
a  3
dtype: int64
```

The returned dtype of the grouped will always include all of the categories that were grouped.

```python
In [178]: s = (pd.Series([1, 1, 1]).groupby(pd.Categorical(["a", "a", "a"], categories=["a", "b"]),
    ....:     observed=False)
    ....:     .count() )
    ....:
In [179]: s.index.dtype
Out[179]: CategoricalDtype(categories=['a', 'b'], ordered=False)
```
NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. In other words, there will never be an “NA group” or “NaT group”. This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

Grouping with ordered factors

Categorical variables represented as instance of pandas’s Categorical class can be used as group keys. If so, the order of the levels will be preserved:

```python
In [180]: data = pd.Series(np.random.randn(100))
In [181]: factor = pd.qcut(data, 
    ...: [0, 0.25, 0.5, 0.75, 1.0])
In [182]: data.groupby(factor).mean()
Out[182]:
      -2.645  -0.523  0.361802  1.073801  
(-2.645, -0.523] -1.362896     0.361802     1.073801  
(-0.523, 0.0296] -0.260266     0.361802     1.073801  
(0.0296, 0.654]  0.361802     0.361802     1.073801  
(0.654, 2.21]   1.073801     1.073801     1.073801  
dtype: float64
```

Grouping with a grouper specification

You may need to specify a bit more data to properly group. You can use the pd.Grouper to provide this local control.

```python
In [183]: import datetime
In [184]: df = pd.DataFrame(
    ...:     {
    ...:         "Branch": "A A A A A A A B".split(),
    ...:         "Buyer": "Carl Mark Carl Carl Joe Joe Joe Carl".split(),
    ...:         "Quantity": [1, 3, 5, 1, 8, 1, 9, 3],
    ...:         "Date": [datetime.datetime(2013, 1, 1, 13, 0),
    ...:                    datetime.datetime(2013, 1, 1, 13, 5),
    ...:                    datetime.datetime(2013, 1, 1, 13, 3),
    ...:                    datetime.datetime(2013, 10, 1, 20, 0),
    ...:                    datetime.datetime(2013, 10, 2, 10, 0),
    ...:                    datetime.datetime(2013, 10, 1, 20, 0),
    ...:                    datetime.datetime(2013, 10, 2, 10, 0),
    ...:                    datetime.datetime(2013, 12, 2, 12, 0),
    ...:                    datetime.datetime(2013, 12, 2, 14, 0),
    ...:                    ],
    ...:     })
In [185]: df
Out[185]:
    Branch  Buyer  Quantity  Date
0   A  Carl       1  2013-01-01 13:00:00
```

2.18. Group by: split-apply-combine 811
Groupby a specific column with the desired frequency. This is like resampling.

```python
In [186]: df.groupby([pd.Grouper(freq="1M", key="Date"), "Buyer"]).sum()
Out[186]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-31</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2013-10-31</td>
<td>Carl</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>9</td>
</tr>
<tr>
<td>2013-12-31</td>
<td>Carl</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>9</td>
</tr>
</tbody>
</table>
```

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

```python
In [187]: df = df.set_index("Date")

In [188]: df["Date"] = df.index + pd.offsets.MonthEnd(2)

In [189]: df.groupby([pd.Grouper(freq="6M", key="Date"), "Buyer"]).sum()
Out[189]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-02-28</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2014-02-28</td>
<td>Carl</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>18</td>
</tr>
</tbody>
</table>

In [190]: df.groupby([pd.Grouper(freq="6M", level="Date"), "Buyer"]).sum()
Out[190]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Buyer</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-31</td>
<td>Carl</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>3</td>
</tr>
<tr>
<td>2014-01-31</td>
<td>Carl</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>18</td>
</tr>
</tbody>
</table>
Taking the first rows of each group

Just like for a DataFrame or Series you can call head and tail on a groupby:

```python
In [191]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [192]: df
Out[192]:
   A  B
0  1  2
1  1  4
2  5  6

In [193]: g = df.groupby('A')

In [194]: g.head(1)
Out[194]:
   A  B
0  1  2
2  5  6

In [195]: g.tail(1)
Out[195]:
   A  B
1  1  4
2  5  6
```

This shows the first or last n rows from each group.

Taking the nth row of each group

To select from a DataFrame or Series the nth item, use `nth()`. This is a reduction method, and will return a single row (or no row) per group if you pass an int for n:

```python
In [196]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [197]: g = df.groupby('A')

In [198]: g.nth(0)
Out[198]:
   B
A
  1  NaN
  5  6.0

In [199]: g.nth(-1)
Out[199]:
   B
A
  1  4.0
  5  6.0

In [200]: g.nth(1)
```

(continues on next page)
If you want to select the nth not-null item, use the `dropna` kwarg. For a DataFrame this should be either 'any' or 'all' just like you would pass to `dropna`:

```
# nth(0) is the same as g.first()
In [201]: g.nth(0, dropna="any")
Out[201]:
   B
A  
1  4.0
5  6.0

In [202]: g.first()
Out[202]:
   B
A  
1  4.0
5  6.0

# nth(-1) is the same as g.last()
In [203]: g.nth(-1, dropna="any")  # NaNs denote group exhausted when using dropna
Out[203]:
   B
A  
1  4.0
5  6.0

In [204]: g.last()
Out[204]:
   B
A  
1  4.0
5  6.0

In [205]: g.B.nth(0, dropna="all")
Out[205]:

   Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```
In [206]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=["A", "B"])
In [207]: g = df.groupby("A", as_index=False)
In [208]: g.nth(0)
```
Out[208]:
     A  B
0  NaN
2  5  6.0

In[209]: g.nth(-1)
Out[209]:
     A  B
0  1  4.0
2  5  6.0

You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

In[210]: business_dates = pd.date_range(start="4/1/2014", end="6/30/2014", freq="B")
In[211]: df = pd.DataFrame(1, index=business_dates, columns=["a", "b"])

# get the first, 4th, and last date index for each month
In[212]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[212]:
     a  b
2014  4  1  1
      4  1  1
      5  1  1
      5  1  1
      6  1  1
      6  1  1

Enumerate group items

To see the order in which each row appears within its group, use the cumcount method:

In[213]: dfg = pd.DataFrame(list("aaabba"), columns=["A"])
In[214]: dfg
Out[214]:
     A
0  a
1  a
2  a
3  b
4  b
5  a

In[215]: dfg.groupby("A").cumcount()
Out[215]:
     0  1
0  0  1
1  1  1
Enumerate groups

To see the ordering of the groups (as opposed to the order of rows within a group given by `cumcount`) you can use `ngroup()`.

Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the `groupby` object, not the order they are first observed.

```python
In [216]: dfg.groupby("A").cumcount(ascending=False)
Out[216]:
       0   1   2
0   0   1   2
1   3   2   1
2   1   4   0
3   0   5   3
dtype: int64
```

```python
In [217]: dfg = pd.DataFrame(list("aabba"), columns=["A"])
In [218]: dfg
Out[218]:
   A
0  a
1  a
2  a
3  b
4  b
5  a
In [219]: dfg.groupby("A").ngroup()
Out[219]:
   0   1   2
0  0   0   0
1  0   1   0
2  0   3   1
3  1   4   1
4  0   5   0
dtype: int64
```

```python
In [220]: dfg.groupby("A").ngroup(ascending=False)
Out[220]:
   0   1
0  1   1
1  2   1
```

(continues on next page)
Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is “B” are 3 higher on average.

```python
In [221]: np.random.seed(1234)
In [222]: df = pd.DataFrame(np.random.randn(50, 2))
In [223]: df["g"] = np.random.choice(["A", "B"], size=50)
In [224]: df.loc[df["g"] == "B", 1] += 3
We can easily visualize this with a boxplot:
```
```python
In [225]: df.groupby("g").boxplot()
Out[225]:
A AxesSubplot(0.1,0.15;0.363636x0.75)
B AxesSubplot(0.536364,0.15;0.363636x0.75)
dtype: object
```
The result of calling `boxplot` is a dictionary whose keys are the values of our grouping column `g` (“A” and “B”). The values of the resulting dictionary can be controlled by the `return_type` keyword of `boxplot`. See the visualization documentation for more.

**Warning:** For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See [here](#) for an explanation.

## Piping function calls

Similar to the functionality provided by `DataFrame` and `Series`, functions that take `GroupBy` objects can be chained together using a `pipe` method to allow for a cleaner, more readable syntax. To read about `.pipe` in general terms, see [here](#).

Combining `.groupby` and `.pipe` is often useful when you need to reuse `GroupBy` objects.

As an example, imagine having a DataFrame with columns for stores, products, revenue and quantity sold. We’d like to do a groupwise calculation of `prices` (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable. First we set the data:

```python
In [226]: n = 1000
In [227]: df = pd.DataFrame(
```

(continues on next page)
In [228]: df.head(2)
Out[228]:
<table>
<thead>
<tr>
<th>Store</th>
<th>Product</th>
<th>Revenue</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store_2</td>
<td>Product_1</td>
<td>26.12</td>
<td>1</td>
</tr>
<tr>
<td>Store_2</td>
<td>Product_1</td>
<td>28.86</td>
<td>1</td>
</tr>
</tbody>
</table>

Now, to find prices per store/product, we can simply do:

In [229]:
(......
   df.groupby(["Store", "Product"])
   .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   .unstack()
   .round(2)
   )
Out[229]:
<table>
<thead>
<tr>
<th>Product</th>
<th>Product_1</th>
<th>Product_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store</td>
<td>Store_1</td>
<td>6.82</td>
</tr>
<tr>
<td></td>
<td>Store_2</td>
<td>6.30</td>
</tr>
<tr>
<td>Store_1</td>
<td>Product_1</td>
<td>7.05</td>
</tr>
<tr>
<td>Store_2</td>
<td>Product_1</td>
<td>6.64</td>
</tr>
</tbody>
</table>

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

In [230]: def mean(groupby):
   ....:     return groupby.mean()
   ....:
In [231]: df.groupby(["Store", "Product"]).pipe(mean)
Out[231]:
<table>
<thead>
<tr>
<th>Store</th>
<th>Product</th>
<th>Revenue</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store_1</td>
<td>Product_1</td>
<td>34.622727</td>
<td>5.075758</td>
</tr>
<tr>
<td></td>
<td>Product_2</td>
<td>35.482815</td>
<td>5.029630</td>
</tr>
<tr>
<td>Store_2</td>
<td>Product_1</td>
<td>32.972837</td>
<td>5.237589</td>
</tr>
<tr>
<td></td>
<td>Product_2</td>
<td>34.684360</td>
<td>5.224000</td>
</tr>
</tbody>
</table>

where mean takes a GroupBy object and finds the mean of the Revenue and Quantity columns respectively for each Store-Product combination. The mean function can be any function that takes in a GroupBy object; the .pipe will pass the GroupBy object as a parameter into the function you specify.
2.18.11 Examples

Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

```python
In [232]: df = pd.DataFrame({"a": [1, 0, 0], "b": [0, 1, 0], "c": [1, 0, 0], "d": [2, 3, 4]})
In [233]: df
Out[233]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4
In [234]: df.groupby(df.sum(), axis=1).sum()
Out[234]:
   1  9
0  2  2
1  1  3
2  0  4
```

Multi-column factorization

By using `ngroup()`, we can extract information about the groups in a way similar to `factorize()` (as described further in the reshaping API) but which applies naturally to multiple columns of mixed type and different sources. This can be useful as an intermediate categorical-like step in processing, when the relationships between the group rows are more important than their content, or as input to an algorithm which only accepts the integer encoding. (For more information about support in pandas for full categorical data, see the Categorical introduction and the API documentation.)

```python
In [235]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})
In [236]: dfg
Out[236]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a
In [237]: dfg.groupby(["A", "B"]).ngroup()
Out[237]:
   0  0
0  1  1
1  2  1
2  3  2
3  4  1
dtype: int64
In [238]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
```

(continues on next page)
Groupby by indexer to ‘resample’ data

Resampling produces new hypothetical samples (resamples) from already existing observed data or from a model that generates data. These new samples are similar to the pre-existing samples.

In order to resample to work on indices that are non-datetimelike, the following procedure can be utilized.

In the following examples, `df.index // 5` returns a binary array which is used to determine what gets selected for the `groupby` operation.

Note: The below example shows how we can downsample by consolidation of samples into fewer samples. Here by using `df.index // 5`, we are aggregating the samples in bins. By applying `std()` function, we aggregate the information contained in many samples into a small subset of values which is their standard deviation thereby reducing the number of samples.

```
In [239]: df = pd.DataFrame(np.random.randn(10, 2))

In [240]: df
Out[240]:
     0      1
0 -0.793893  0.321153
1  0.342250  1.618906
2 -0.975807  1.918201
3 -0.810847 -1.405919
4 -1.977759  0.461659
5  0.730057 -1.316938
6  0.730057  1.316938
7 -0.505895 -1.081009
8  0.505895 -1.701948
9 -1.006349  0.020208

In [241]: df.index // 5
Out[241]: Int64Index([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype='int64')

In [242]: df.groupby(df.index // 5).std()
Out[242]:
     0      1
0  0.823647  1.312912
1  0.760109  0.942941
```
Returning a Series to propagate names

Group DataFrame columns, compute a set of metrics and return a named Series. The Series name is used as the name for the column index. This is especially useful in conjunction with reshaping operations such as stacking in which the column index name will be used as the name of the inserted column:

```python
In [243]: df = pd.DataFrame(
......:     {
......:         "a": [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2],
......:         "b": [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
......:         "c": [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
......:         "d": [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
......:     }
......: )
......:
In [244]: def compute_metrics(x):
......:     result = {"b_sum": x["b"].sum(), "c_mean": x["c"].mean()}
......:     return pd.Series(result, name="metrics")
......:
In [245]: result = df.groupby("a").apply(compute_metrics)
```

```
In [246]: result
Out[246]:
metrics  b_sum  c_mean
a
0   2.0   0.5
1   2.0   0.5
2   2.0   0.5
```

```
In [247]: result.stack()
Out[247]:
        a  metrics
    b_sum  c_mean
0     b_sum  2.0  0.5
       c_mean  0.5
1     b_sum  2.0  0.5
       c_mean  0.5
2     b_sum  2.0  0.5
       c_mean  0.5
dtype: float64
```

2.19 Windowing Operations

pandas contains a compact set of APIs for performing windowing operations - an operation that performs an aggregation over a sliding partition of values. The API functions similarly to the `groupby` API in that `Series` and `DataFrame` call the windowing method with necessary parameters and then subsequently call the aggregation function.

```
In [1]: s = pd.Series(range(5))
In [2]: s.rolling(window=2).sum()
(continues on next page)
The windows are comprised by looking back the length of the window from the current observation. The result above can be derived by taking the sum of the following windowed partitions of data:

```
In [3]: for window in s.rolling(window=2):
    ...:     print(window)
    ...:
0  0
0  0
dtype: int64
0  0
1  1
dtype: int64
1  1
2  2
dtype: int64
2  2
3  3
dtype: int64
3  3
4  4
dtype: int64
```

2.19.1 Overview

pandas supports 4 types of windowing operations:

1. Rolling window: Generic fixed or variable sliding window over the values.
2. Weighted window: Weighted, non-rectangular window supplied by the `scipy.signal` library.
3. Expanding window: Accumulating window over the values.
4. Exponentially Weighted window: Accumulating and exponentially weighted window over the values.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Method</th>
<th>Returned Object</th>
<th>Supports time-based windows</th>
<th>Supports chained groupby</th>
<th>Supports table method</th>
<th>Supports online operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rolling window</td>
<td>rolling</td>
<td>Rolling</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (as of version 1.3)</td>
<td>No</td>
</tr>
<tr>
<td>Weighted window</td>
<td>rolling</td>
<td>Window</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Expanding window</td>
<td>expanding</td>
<td>Expanding</td>
<td>No</td>
<td>Yes</td>
<td>Yes (as of version 1.3)</td>
<td>No</td>
</tr>
<tr>
<td>Exponentially Weighted window</td>
<td>ewm</td>
<td>ExponentialMovingWindow</td>
<td>Yes (as of version 1.2)</td>
<td>No</td>
<td>Yes (as of version 1.3)</td>
<td>No</td>
</tr>
</tbody>
</table>
As noted above, some operations support specifying a window based on a time offset:

```python
In [4]: s = pd.Series(range(5), index=pd.date_range('2020-01-01', periods=5, freq='1D'))
In [5]: s.rolling(window='2D').sum()
Out[5]:
2020-01-01 0.0
2020-01-02 1.0
2020-01-03 3.0
2020-01-04 5.0
2020-01-05 7.0
Freq: D, dtype: float64
```

Additionally, some methods support chaining a `groupby` operation with a windowing operation which will first group the data by the specified keys and then perform a windowing operation per group.

```python
In [6]: df = pd.DataFrame({'A': ['a', 'b', 'a', 'b', 'a'], 'B': range(5)})
In [7]: df.groupby('A').expanding().sum()
Out[7]:
   B
A
a  0  0.0
   2  2.0
   4  6.0
b  1  1.0
   3  4.0
```

**Note:** Windowing operations currently only support numeric data (integer and float) and will always return `float64` values.

**Warning:** Some windowing aggregation, `mean`, `sum`, `var` and `std` methods may suffer from numerical imprecision due to the underlying windowing algorithms accumulating sums. When values differ with magnitude $1/\text{np.finfo(np.double).eps}$ this results in truncation. It must be noted, that large values may have an impact on windows, which do not include these values. Kahan summation is used to compute the rolling sums to preserve accuracy as much as possible.

New in version 1.3.0.

Some windowing operations also support the `method='table'` option in the constructor which performs the windowing operation over an entire `DataFrame` instead of a single column or row at a time. This can provide a useful performance benefit for a `DataFrame` with many columns or rows (with the corresponding `axis` argument) or the ability to utilize other columns during the windowing operation. The `method='table'` option can only be used if `engine='numba'` is specified in the corresponding method call.

For example, a weighted mean calculation can be calculated with `apply()` by specifying a separate column of weights.

```python
In [8]: def weighted_mean(x):
   ...:     arr = np.ones((1, x.shape[1]))
   ...:     arr[:, :2] = (x[:, :2] * x[:, 2]).sum(axis=0) / x[:, 2].sum()
   ...:     return arr
```

(continues on next page)
...:

In [9]: df = pd.DataFrame([[1, 2, 0.6], [2, 3, 0.4], [3, 4, 0.2], [4, 5, 0.7]])

In [10]: df.rolling(2, method="table", min_periods=0).apply(weighted_mean, raw=True, engine="numba") # noqa:E501
Out[10]:
    0         1         2
0  1.000000  2.000000  1.0
1  1.800000  2.000000  1.0
2  3.333333  2.333333  1.0
3  1.555556  7.000000  1.0

New in version 1.3.

Some windowing operations also support an online method after constructing a windowing object which returns a new object that supports passing in new DataFrame or Series objects to continue the windowing calculation with the new values (i.e. online calculations).

The methods on this new windowing objects must call the aggregation method first to “prime” the initial state of the online calculation. Then, new DataFrame or Series objects can be passed in the update argument to continue the windowing calculation.

In [11]: df = pd.DataFrame([[1, 2, 0.6], [2, 3, 0.4], [3, 4, 0.2], [4, 5, 0.7]])

In [12]: df.ewm(0.5).mean()
Out[12]:
    0         1         2
0  1.000000  2.000000  0.600000
1  1.750000  2.750000  0.450000
2  2.615385  3.615385  0.276923
3  3.550000  4.550000  0.562500

In [13]: online_ewm = df.head(2).ewm(0.5).online()

In [14]: online_ewm.mean()
Out[14]:
    0         1         2
0  1.00  2.00  0.60
1  1.75  2.75  0.45

In [15]: online_ewm.mean(update=df.tail(1))
Out[15]:
    0         1         2
3  3.307692  4.307692  0.623077

All windowing operations support a min_periods argument that dictates the minimum amount of non-np.nan values a window must have; otherwise, the resulting value is np.nan. min_periods defaults to 1 for time-based windows and window for fixed windows.

In [16]: s = pd.Series([np.nan, 1, 2, np.nan, np.nan, 3])

In [17]: s.rolling(window=3, min_periods=1).sum()
Additionally, all windowing operations support the `aggregate` method for returning a result of multiple aggregations applied to a window.
2.19.2 Rolling window

Generic rolling windows support specifying windows as a fixed number of observations or variable number of observations based on an offset. If a time based offset is provided, the corresponding time based index must be monotonic.

```
In [22]: times = ['2020-01-01', '2020-01-03', '2020-01-04', '2020-01-05', '2020-01-29']

In [23]: s = pd.Series(range(5), index=pd.DatetimeIndex(times))

In [24]: s
Out[24]:
2020-01-01 0
2020-01-03 1
2020-01-04 2
2020-01-05 3
2020-01-29 4
dtype: int64

# Window with 2 observations
In [25]: s.rolling(window=2).sum()
Out[25]:
2020-01-01 NaN
2020-01-03 1.0
2020-01-04 3.0
2020-01-05 5.0
2020-01-29 7.0
dtype: float64

# Window with 2 days worth of observations
In [26]: s.rolling(window='2D').sum()
Out[26]:
2020-01-01 0.0
2020-01-03 1.0
2020-01-04 3.0
2020-01-05 5.0
2020-01-29 4.0
dtype: float64
```

For all supported aggregation functions, see `Rolling window functions`.

Centering windows

By default the labels are set to the right edge of the window, but a `center` keyword is available so the labels can be set at the center.

```
In [27]: s = pd.Series(range(10))

In [28]: s.rolling(window=5).mean()
Out[28]:
0   NaN
1   NaN
2   NaN
3   NaN
```

(continues on next page)
This can also be applied to datetime-like indices.

New in version 1.3.0.

```
In [30]: df = pd.DataFrame(
        ....:     {"A": [0, 1, 2, 3, 4]}, index=pd.date_range("2020", periods=5, freq="1D")
        ....:     )
        
In [31]: df
Out[31]:
   A
2020-01-01  0
2020-01-02  1
2020-01-03  2
2020-01-04  3
2020-01-05  4

In [32]: df.rolling("2D", center=False).mean()
Out[32]:
   A
2020-01-01  0.0
2020-01-02  0.5
2020-01-03  1.5
2020-01-04  2.5
2020-01-05  3.5

In [33]: df.rolling("2D", center=True).mean()
Out[33]:
   A
2020-01-01  0.5
```
Rolling window endpoints

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

<table>
<thead>
<tr>
<th>Value</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>'right'</td>
<td>close right endpoint</td>
</tr>
<tr>
<td>'left'</td>
<td>close left endpoint</td>
</tr>
<tr>
<td>'both'</td>
<td>close both endpoints</td>
</tr>
<tr>
<td>'neither'</td>
<td>open endpoints</td>
</tr>
</tbody>
</table>

For example, having the right endpoint open is useful in many problems that require that there is no contamination from present information back to past information. This allows the rolling window to compute statistics “up to that point in time”, but not including that point in time.

```python
In [34]: df = pd.DataFrame(    ....:    {"x": 1},
    ....:    index=[    ....:        pd.Timestamp("20130101 09:00:01"),
    ....:        pd.Timestamp("20130101 09:00:02"),
    ....:        pd.Timestamp("20130101 09:00:03"),
    ....:        pd.Timestamp("20130101 09:00:04"),
    ....:        pd.Timestamp("20130101 09:00:06"),
    ....:    ],
    ....: )

In [35]: df["right"] = df.rolling("2s", closed="right").x.sum() # default

In [36]: df["both"] = df.rolling("2s", closed="both").x.sum()

In [37]: df["left"] = df.rolling("2s", closed="left").x.sum()

In [38]: df["neither"] = df.rolling("2s", closed="neither").x.sum()

In [39]: df
Out[39]:
```

<table>
<thead>
<tr>
<th></th>
<th>right</th>
<th>both</th>
<th>left</th>
<th>neither</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 09:00:01</td>
<td>1.0</td>
<td>1.0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-01 09:00:02</td>
<td>2.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:03</td>
<td>2.0</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:04</td>
<td>2.0</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:06</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Custom window rolling

New in version 1.0.

In addition to accepting an integer or offset as a window argument, rolling also accepts a BaseIndexer subclass that allows a user to define a custom method for calculating window bounds. The BaseIndexer subclass will need to define a get_window_bounds method that returns a tuple of two arrays, the first being the starting indices of the windows and second being the ending indices of the windows. Additionally, num_values, min_periods, center, closed and will automatically be passed to get_window_bounds and the defined method must always accept these arguments.

For example, if we have the following DataFrame

```
In [40]: use_expanding = [True, False, True, False, True]

In [41]: use_expanding
Out[41]: [True, False, True, False, True]

In [42]: df = pd.DataFrame({"values": range(5)})

In [43]: df
Out[43]:
   values
0      0
1      1
2      2
3      3
4      4
```

...and we want to use an expanding window where use_expanding is True otherwise a window of size 1, we can create the following BaseIndexer subclass:

```
In [2]: from pandas.api.indexers import BaseIndexer

In [3]: class CustomIndexer(BaseIndexer):
   ...:     def get_window_bounds(self, num_values, min_periods, center, closed):
   ...:         start = np.empty(num_values, dtype=np.int64)
   ...:         end = np.empty(num_values, dtype=np.int64)
   ...:         for i in range(num_values):
   ...:             if self.use_expanding[i]:
   ...:                 start[i] = 0
   ...:                 end[i] = i + 1
   ...:             else:
   ...:                 start[i] = i
   ...:                 end[i] = i + self.window_size
   ...:         return start, end

In [4]: indexer = CustomIndexer(window_size=1, use_expanding=use_expanding)

In [5]: df.rolling(indexer).sum()
Out[5]:
   values
0      0.0
1      1.0
2      3.0
```

(continues on next page)
You can view other examples of `BaseIndexer` subclasses here.

New in version 1.1.

One subclass of note within those examples is the `VariableOffsetWindowIndexer` that allows rolling operations over a non-fixed offset like a `BusinessDay`.

```python
In [44]: from pandas.api.indexers import VariableOffsetWindowIndexer

In [45]: df = pd.DataFrame(range(10), index=pd.date_range("2020", periods=10))

In [46]: offset = pd.offsets.BDay(1)

In [47]: indexer = VariableOffsetWindowIndexer(index=df.index, offset=offset)

In [48]: df
Out[48]:
   0
2020-01-01 0
2020-01-02 1
2020-01-03 2
2020-01-04 3
2020-01-05 4
2020-01-06 5
2020-01-07 6
2020-01-08 7
2020-01-09 8
2020-01-10 9

In [49]: df.rolling(indexer).sum()
Out[49]:
   0
2020-01-01 0.0
2020-01-02 1.0
2020-01-03 2.0
2020-01-04 3.0
2020-01-05 7.0
2020-01-06 12.0
2020-01-07 6.0
2020-01-08 7.0
2020-01-09 8.0
2020-01-10 9.0
```

For some problems knowledge of the future is available for analysis. For example, this occurs when each data point is a full time series read from an experiment, and the task is to extract underlying conditions. In these cases it can be useful to perform forward-looking rolling window computations. `FixedForwardWindowIndexer` class is available for this purpose. This `BaseIndexer` subclass implements a closed fixed-width forward-looking rolling window, and we can use it as follows:

```python
In [50]: from pandas.api.indexers import FixedForwardWindowIndexer
```

(continues on next page)
In [51]: indexer = FixedForwardWindowIndexer(window_size=2)

In [52]: df.rolling(indexer, min_periods=1).sum()

Out[52]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-01</td>
<td>1.0</td>
</tr>
<tr>
<td>2020-01-02</td>
<td>3.0</td>
</tr>
<tr>
<td>2020-01-03</td>
<td>5.0</td>
</tr>
<tr>
<td>2020-01-04</td>
<td>7.0</td>
</tr>
<tr>
<td>2020-01-05</td>
<td>9.0</td>
</tr>
<tr>
<td>2020-01-06</td>
<td>11.0</td>
</tr>
<tr>
<td>2020-01-07</td>
<td>13.0</td>
</tr>
<tr>
<td>2020-01-08</td>
<td>15.0</td>
</tr>
<tr>
<td>2020-01-09</td>
<td>17.0</td>
</tr>
<tr>
<td>2020-01-10</td>
<td>9.0</td>
</tr>
</tbody>
</table>

We can also achieve this by using slicing, applying rolling aggregation, and then flipping the result as shown in example below:

In [53]: df = pd.DataFrame(
    ....:     data=[
    ....:         [pd.Timestamp("2018-01-01 00:00:00"), 100],
    ....:         [pd.Timestamp("2018-01-01 00:00:01"), 101],
    ....:         [pd.Timestamp("2018-01-01 00:00:03"), 103],
    ....:         [pd.Timestamp("2018-01-01 00:00:04"), 111],
    ....:     ],
    ....:     columns=["time", "value"],
    ....: ).set_index("time")
    ....: 

In [54]: df

Out[54]:

<table>
<thead>
<tr>
<th>time</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-01 00:00:00</td>
<td>100</td>
</tr>
<tr>
<td>2018-01-01 00:00:01</td>
<td>101</td>
</tr>
<tr>
<td>2018-01-01 00:00:03</td>
<td>103</td>
</tr>
<tr>
<td>2018-01-01 00:00:04</td>
<td>111</td>
</tr>
</tbody>
</table>

In [55]: reversed_df = df[::-1].rolling("2s").sum()[::-1]

In [56]: reversed_df

Out[56]:

<table>
<thead>
<tr>
<th>time</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-01 00:00:00</td>
<td>201.0</td>
</tr>
<tr>
<td>2018-01-01 00:00:01</td>
<td>101.0</td>
</tr>
<tr>
<td>2018-01-01 00:00:03</td>
<td>214.0</td>
</tr>
<tr>
<td>2018-01-01 00:00:04</td>
<td>111.0</td>
</tr>
</tbody>
</table>
Rolling apply

The apply() function takes an extra func argument and performs generic rolling computations. The func argument should be a single function that produces a single value from an ndarray input. raw specifies whether the windows are cast as Series objects (raw=False) or ndarray objects (raw=True).

```python
In [57]: def mad(x):
       ...:     return np.fabs(x - x.mean()).mean()
       ....:

In [58]: s = pd.Series(range(10))

In [59]: s.rolling(window=4).apply(mad, raw=True)
Out[59]:
0    NaN
1    NaN
2    NaN
3    1.0
4    1.0
5    1.0
6    1.0
7    1.0
8    1.0
9    1.0
dtype: float64
```

Numba engine

New in version 1.0.

Additionally, apply() can leverage Numba if installed as an optional dependency. The apply aggregation can be executed using Numba by specifying engine='numba' and engine_kwargs arguments (raw must also be set to True). See enhancing performance with Numba for general usage of the arguments and performance considerations.

Numba will be applied in potentially two routines:

1. If func is a standard Python function, the engine will JIT the passed function. func can also be a JITed function in which case the engine will not JIT the function again.

2. The engine will JIT the for loop where the apply function is applied to each window.

The engine_kwargs argument is a dictionary of keyword arguments that will be passed into the numba.jit decorator. These keyword arguments will be applied to both the passed function (if a standard Python function) and the apply for loop over each window.

New in version 1.3.0.

mean, median, max, min, and sum also support the engine and engine_kwargs arguments.
Binary window functions

cov() and corr() can compute moving window statistics about two Series or any combination of
DataFrame/Series or DataFrame/DataFrame. Here is the behavior in each case:

- two Series: compute the statistic for the pairing.
- DataFrame/Series: compute the statistics for each column of the DataFrame with the passed Series, thus
  returning a DataFrame.
- DataFrame/DataFrame: by default compute the statistic for matching column names, returning a DataFrame. If
  the keyword argument pairwise=True is passed then computes the statistic for each pair of columns, returning
  a DataFrame with a MultiIndex whose values are the dates in question (see the next section).

For example:

```python
In [60]: df = pd.DataFrame(
    ....:     np.random.randn(10, 4),
    ....:     index=pd.date_range("2020-01-01", periods=10),
    ....:     columns=["A", "B", "C", "D"],
    ....: )

In [61]: df = df.cumsum()

In [62]: df2 = df[:4]

In [63]: df2.rolling(window=2).corr(df2["B"])

Out[63]:
         A   B     C   D
2020-01-01 NaN NaN NaN NaN
2020-01-02 -1.0 1.0 -1.0 1.0
2020-01-03  1.0 1.0  1.0 -1.0
2020-01-04 -1.0 1.0  1.0 -1.0
```

Computing rolling pairwise covariances and correlations

In financial data analysis and other fields it’s common to compute covariance and correlation matrices for a collection
of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by
passing the pairwise keyword argument, which in the case of DataFrame inputs will yield a MultiIndexed DataFrame
whose index are the dates in question. In the case of a single DataFrame argument the pairwise argument can even be omitted:

```python
In [64]: covs = (  
    ....:     df["B", "C", "D"]
    ....:     .rolling(window=4)
    ....:     .cov(df["A", "B", "C"], pairwise=True)
    ....: )
```

Note: Missing values are ignored and each entry is computed using the pairwise complete observations. Please see
the covariance section for caveats associated with this method of calculating covariance and correlation matrices.
In [65]: covs
Out[65]:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-01</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2020-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2020-01-09</td>
<td>0.342006</td>
<td>0.230190</td>
<td>0.052849</td>
</tr>
<tr>
<td></td>
<td>0.230190</td>
<td>1.575251</td>
<td>0.082901</td>
</tr>
<tr>
<td>2020-01-10</td>
<td>-0.333945</td>
<td>0.006871</td>
<td>-0.655514</td>
</tr>
<tr>
<td></td>
<td>0.649711</td>
<td>0.430860</td>
<td>0.469271</td>
</tr>
<tr>
<td></td>
<td>0.430860</td>
<td>0.829721</td>
<td>0.055300</td>
</tr>
</tbody>
</table>

[30 rows x 3 columns]

### 2.19.3 Weighted window

The `win_type` argument in `.rolling` generates a weighted windows that are commonly used in filtering and spectral estimation. `win_type` must be string that corresponds to a `scipy.signal` window function. Scipy must be installed in order to use these windows, and supplementary arguments that the Scipy window methods take must be specified in the aggregation function.

In [66]: s = pd.Series(range(10))

In [67]: s.rolling(window=5).mean()
Out[67]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>2.0</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>4.0</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>5.0</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>6.0</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>7.0</td>
<td></td>
</tr>
</tbody>
</table>
| dtype: float64

In [68]: s.rolling(window=5, win_type="triang").mean()
Out[68]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>2.0</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>4.0</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>5.0</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>6.0</td>
<td></td>
</tr>
</tbody>
</table>
# Supplementary Scipy arguments passed in the aggregation function

In [69]: s.rolling(window=5, win_type="gaussian").mean(std=0.1)

Out[69]:
0    NaN
1    NaN
2    NaN
3    NaN
4    2.0
5    3.0
6    4.0
7    5.0
8    6.0
9    7.0
dtype: float64

For all supported aggregation functions, see [Weighted window functions](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rolling.html).

## 2.19.4 Expanding window

An expanding window yields the value of an aggregation statistic with all the data available up to that point in time. Since these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

In [70]: df = pd.DataFrame(range(5))

In [71]: df.rolling(window=len(df), min_periods=1).mean()

Out[71]:
0   0.0
1   0.5
2   1.0
3   1.5
4   2.0

In [72]: df.expanding(min_periods=1).mean()

Out[72]:
0   0.0
1   0.5
2   1.0
3   1.5
4   2.0

For all supported aggregation functions, see [Expanding window functions](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.expanding.html).
2.19.5 Exponentially Weighted window

An exponentially weighted window is similar to an expanding window but with each prior point being exponentially weighted down relative to the current point.

In general, a weighted moving average is calculated as

\[ y_t = \frac{\sum_{i=0}^{t} w_i x_{t-i}}{\sum_{i=0}^{t} w_i}, \]

where \( x_t \) is the input, \( y_t \) is the result and the \( w_i \) are the weights.

For all supported aggregation functions, see Exponentially-weighted window functions.

The EW functions support two variants of exponential weights. The default, \( \text{adjust=True} \), uses the weights \( w_i = (1 - \alpha)^i \) which gives

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots + (1 - \alpha)^ix_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots + (1 - \alpha)^t}. \]

When \( \text{adjust=False} \) is specified, moving averages are calculated as

\[ y_0 = x_0 \]
\[ y_t = (1 - \alpha)y_{t-1} + \alpha x_t, \]

which is equivalent to using weights

\[ w_i = \begin{cases} \alpha(1 - \alpha)^i & \text{if } i < t \\ (1 - \alpha)^i & \text{if } i = t. \end{cases} \]

**Note:** These equations are sometimes written in terms of \( \alpha' = 1 - \alpha \), e.g.

\[ y_t = \alpha'y_{t-1} + (1 - \alpha')x_t. \]

The difference between the above two variants arises because we are dealing with series which have finite history. Consider a series of infinite history, with \( \text{adjust=True} \):

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots} \]

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of 1 - \( \alpha \) we have

\[ y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots}{1 - (1 - \alpha)} \]
\[ = \frac{(x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots)\alpha}{\alpha} \]
\[ = \frac{\alpha x_t + [(1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots]\alpha}{\alpha} \]
\[ = \alpha x_t + (1 - \alpha)[x_{t-1} + (1 - \alpha)x_{t-2} + \ldots] \]
\[ = \alpha x_t + (1 - \alpha)y_{t-1} \]

which is the same expression as \( \text{adjust=False} \) above and therefore shows the equivalence of the two variants for infinite series. When \( \text{adjust=False} \), we have \( y_0 = x_0 \) and \( y_t = \alpha x_t + (1 - \alpha)y_{t-1} \). Therefore, there is an assumption that \( x_0 \) is not an ordinary value but rather an exponentially weighted moment of the infinite series up to that point.
One must have $0 < \alpha \leq 1$, and while it is possible to pass $\alpha$ directly, it’s often easier to think about either the **span**, **center of mass** (**com**) or **half-life** of an EW moment:

$$\alpha = \begin{cases} \frac{2}{s+1}, & \text{for span } s \geq 1 \\ \frac{1}{1+c}, & \text{for center of mass } c \geq 0 \\ 1 - \exp^{-\log 0.5 / h}, & \text{for half-life } h > 0 \end{cases}$$

One must specify precisely one of **span**, **center of mass**, **half-life** and **alpha** to the EW functions:

- **Span** corresponds to what is commonly called an “N-day EW moving average”.
- **Center of mass** has a more physical interpretation and can be thought of in terms of span: $c = (s - 1)/2$.
- **Half-life** is the period of time for the exponential weight to reduce to one half.
- **Alpha** specifies the smoothing factor directly.

New in version 1.1.0.

You can also specify **halflife** in terms of a timedelta convertible unit to specify the amount of time it takes for an observation to decay to half its value when also specifying a sequence of **times**.

```
In [73]: df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})

In [74]: df
Out[74]:
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

In [75]: times = ["2020-01-01", "2020-01-03", "2020-01-10", "2020-01-15", "2020-01-17"]

In [76]: df.ewm(halflife="4 days", times=pd.DatetimeIndex(times)).mean()
Out[76]:
   B
0  0.000000
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

The following formula is used to compute exponentially weighted mean with an input vector of times:

$$y_t = \frac{\sum_{i=0}^{t} 0.5^{\frac{t-i}{s}} x_{t-i}}{\sum_{i=0}^{t} 0.5^{\frac{t-i}{s}}}$$

ExponentialMovingWindow also has an **ignore_na** argument, which determines how intermediate null values affect the calculation of the weights. When **ignore_na**=False (the default), weights are calculated based on absolute positions, so that intermediate null values affect the result. When **ignore_na**=True, weights are calculated by ignoring intermediate null values. For example, assuming adjust=True, if **ignore_na**=False, the weighted average of 3, NaN, 5 would be calculated as

$$\frac{(1-\alpha)^2 \cdot 3 + 1 \cdot 5}{(1-\alpha)^2 + 1}.$$
Whereas if `ignore_na=True`, the weighted average would be calculated as

\[
\frac{(1 - \alpha) \cdot 3 + 1 \cdot 5}{(1 - \alpha) + 1}.
\]

The `var()`, `std()`, and `cov()` functions have a `bias` argument, specifying whether the result should contain biased or unbiased statistics. For example, if `bias=True`, `ewmvar(x)` is calculated as `ewmvar(x) = ewma(x)**2 - ewma(x)**2`; whereas if `bias=False` (the default), the biased variance statistics are scaled by debiasing factors

\[
\frac{\left(\sum_{i=0}^t w_i\right)^2}{\left(\sum_{i=0}^t w_i\right)^2 - \sum_{i=0}^t w_i^2}.
\]

(For \(w_i = 1\), this reduces to the usual \(N/(N-1)\) factor, with \(N = t+1\).) See Weighted Sample Variance on Wikipedia for further details.

## 2.20 Time series / date functionality

pandas contains extensive capabilities and features for working with time series data for all domains. Using the NumPy `datetime64` and `timedelta64` dtypes, pandas has consolidated a large number of features from other Python libraries like `scikits.timeseries` as well as created a tremendous amount of new functionality for manipulating time series data.

For example, pandas supports:

### Parsing time series information from various sources and formats

```python
In [1]: import datetime

In [2]: dti = pd.to_datetime(
   ...:    ['1/1/2018', np.datetime64('2018-01-01'), datetime.datetime(2018, 1, 1)]
   ...: )
   ...

In [3]: dti
Out[3]: DatetimeIndex(['2018-01-01', '2018-01-01', '2018-01-01'], dtype='datetime64[ns]', freq=None)
```

Generate sequences of fixed-frequency dates and time spans

```python
In [4]: dti = pd.date_range('2018-01-01', periods=3, freq='H')

In [5]: dti
Out[5]: DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 01:00:00', '2018-01-01 02:00:00'],
   ...: dtype='datetime64[ns]', freq='H')
```

Manipulating and converting date times with timezone information

```python
In [6]: dti = dti.tz_localize('UTC')

In [7]: dti
Out[7]:
```

(continues on next page)
Resampling or converting a time series to a particular frequency

```
In [9]: idx = pd.date_range("2018-01-01", periods=5, freq="H")
In [10]: ts = pd.Series(range(len(idx)), index=idx)
In [11]: ts
Out[11]:
2018-01-01 00:00:00    0
2018-01-01 01:00:00    1
2018-01-01 02:00:00    2
2018-01-01 03:00:00    3
2018-01-01 04:00:00    4
Freq: H, dtype: int64
```

```
In [12]: ts.resample("2H").mean()
Out[12]:
2018-01-01 00:00:00    0.5
2018-01-01 02:00:00    2.5
2018-01-01 04:00:00    4.0
Freq: 2H, dtype: float64
```

Performing date and time arithmetic with absolute or relative time increments

```
In [13]: friday = pd.Timestamp("2018-01-05")
In [14]: friday.day_name()
Out[14]: 'Friday'

# Add 1 day
In [15]: saturday = friday + pd.Timedelta("1 day")
In [16]: saturday.day_name()
Out[16]: 'Saturday'

# Add 1 business day (Friday --> Monday)
In [17]: monday = friday + pd.offsets.BDay()
In [18]: monday.day_name()
Out[18]: 'Monday'
```

pandas provides a relatively compact and self-contained set of tools for performing the above tasks and more.
2.20.1 Overview

pandas captures 4 general time related concepts:

1. Date times: A specific date and time with timezone support. Similar to `datetime.datetime` from the standard library.
2. Time deltas: An absolute time duration. Similar to `datetime.timedelta` from the standard library.
3. Time spans: A span of time defined by a point in time and its associated frequency.
4. Date offsets: A relative time duration that respects calendar arithmetic. Similar to `dateutil.relativedelta`. 

<table>
<thead>
<tr>
<th>Concept</th>
<th>Scalar Class</th>
<th>Array Class</th>
<th>pandas Data Type</th>
<th>Primary Creation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date times</td>
<td>Timestamp</td>
<td>DatetimeIndex</td>
<td>datetime64[ns] or</td>
<td><code>to_datetime</code> or <code>date_range</code></td>
</tr>
<tr>
<td>Time deltas</td>
<td>Timedelta</td>
<td>TimedeltaIndex</td>
<td>timedelta64[ns]</td>
<td></td>
</tr>
<tr>
<td>Time spans</td>
<td>Period</td>
<td>PeriodIndex</td>
<td>period[freq]</td>
<td><code>Period</code> or <code>period_range</code></td>
</tr>
<tr>
<td>Date offsets</td>
<td>DateOffset</td>
<td>None</td>
<td>None</td>
<td><code>DateOffset</code></td>
</tr>
</tbody>
</table>

For time series data, it’s conventional to represent the time component in the index of a `Series` or `DataFrame` so manipulations can be performed with respect to the time element.

```
In [19]: pd.Series(range(3), index=pd.date_range("2000", freq="D", periods=3))
Out[19]:
2000-01-01 0
2000-01-02 1
2000-01-03 2
Freq: D, dtype: int64
```

However, `Series` and `DataFrame` can directly also support the time component as data itself.

```
In [20]: pd.Series(pd.date_range("2000", freq="D", periods=3))
Out[20]:
0  2000-01-01
1  2000-01-02
2  2000-01-03
dtype: datetime64[ns]
```

`Series` and `DataFrame` have extended data type support and functionality for `datetime`, `timedelta` and `Period` data when passed into those constructors. `DateOffset` data however will be stored as object data.

```
In [21]: pd.Series(pd.period_range("1/1/2011", freq="M", periods=3))
Out[21]:
0  2011-01
1  2011-02
2  2011-03
dtype: period[M]
```

```
In [22]: pd.Series([pd.DateOffset(1), pd.DateOffset(2)])
```

(continues on next page)
Lastly, pandas represents null date times, time deltas, and time spans as \texttt{NaT} which is useful for representing missing or null date like values and behaves similar as \texttt{np.nan} does for float data.

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Timestamp(pd.NaT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>NaT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Timedelta(pd.NaT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>NaT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Period(pd.NaT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>NaT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th># Equality acts as np.nan would</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>pd.NaT == pd.NaT</td>
</tr>
<tr>
<td>Out</td>
<td>False</td>
</tr>
</tbody>
</table>

### 2.20.2 Timestamps vs. time spans

Timestamped data is the most basic type of time series data that associates values with points in time. For pandas objects it means using the points in time.

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Timestamp(datetime.datetime(2012, 5, 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Timestamp('2012-05-01 00:00:00')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Timestamp(&quot;2012-05-01&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Timestamp('2012-05-01 00:00:00')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Timestamp(2012, 5, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Timestamp('2012-05-01 00:00:00')</td>
</tr>
</tbody>
</table>

However, in many cases it is more natural to associate things like change variables with a time span instead. The span represented by \texttt{Period} can be specified explicitly, or inferred from datetime string format.

For example:

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Period(&quot;2011-01&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Period('2011-01', 'M')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Period(&quot;2012-05&quot;, freq=&quot;D&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Period('2012-05-01', 'D')</td>
</tr>
</tbody>
</table>
Timestamp and Period can serve as an index. Lists of Timestamp and Period are automatically coerced to DatetimeIndex and PeriodIndex respectively.

```python
In [33]: dates = [
    ....:     pd.Timestamp("2012-05-01"),
    ....:     pd.Timestamp("2012-05-02"),
    ....:     pd.Timestamp("2012-05-03"),
    ....:     ]
    ....:
In [34]: ts = pd.Series(np.random.randn(3), dates)
In [35]: type(ts.index)
Out[35]: pandas.core.indexes.datetimes.DatetimeIndex
In [36]: ts.index
Out[36]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)
In [37]: ts
Out[37]:
2012-05-01   0.469112
2012-05-02  -0.282863
2012-05-03  -1.509059
dtype: float64
In [38]: periods = [pd.Period("2012-01"), pd.Period("2012-02"), pd.Period("2012-03"))
In [39]: ts = pd.Series(np.random.randn(3), periods)
In [40]: type(ts.index)
Out[40]: pandas.core.indexes.period.PeriodIndex
In [41]: ts.index
Out[41]: PeriodIndex(['2012-01', '2012-02', '2012-03'], dtype='period[M]')
In [42]: ts
Out[42]:
2012-01   -1.135632
2012-02    1.212112
2012-03   -0.173215
Freq: M, dtype: float64
```
pandas allows you to capture both representations and convert between them. Under the hood, pandas represents timestamps using instances of Timestamp and sequences of timestamps using instances of DatetimeIndex. For regular time spans, pandas uses Period objects for scalar values and PeriodIndex for sequences of spans. Better support for irregular intervals with arbitrary start and end points are forth-coming in future releases.
### 2.20.3 Converting to timestamps

To convert a `Series` or list-like object of date-like objects e.g. strings, epochs, or a mixture, you can use the `to_datetime` function. When passed a `Series`, this returns a `Series` (with the same index), while a list-like is converted to a `DatetimeIndex`:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
</table>
| `pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))` | 0 2009-07-31  
1 2010-01-10  
2 NaT  
dtype: datetime64[ns] |

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.to_datetime(['2005/11/23', '2010.12.31'])</code></td>
<td>DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None)</td>
</tr>
</tbody>
</table>

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)</code></td>
<td>DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=None)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)</code></td>
<td>DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]', freq=None)</td>
</tr>
</tbody>
</table>

**Warning:** You see in the above example that `dayfirst` isn't strict. If a date can’t be parsed with the day being first it will be parsed as if `dayfirst` were False, and in the case of parsing delimited date strings (e.g. `31-12-2012`) then a warning will also be raised.

If you pass a single string to `to_datetime`, it returns a single `Timestamp`. `Timestamp` can also accept string input, but it doesn't accept string parsing options like `dayfirst` or `format`, so use `to_datetime` if these are required:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.to_datetime('2010/11/12')</code></td>
<td>Timestamp('2010-11-12 00:00:00')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.Timestamp('2010/11/12')</code></td>
<td>Timestamp('2010-11-12 00:00:00')</td>
</tr>
</tbody>
</table>

You can also use the `DatetimeIndex` constructor directly:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
</table>

The string ‘infer’ can be passed in order to set the frequency of the index as the inferred frequency upon creation:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pd.DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], freq=&quot;infer&quot;)</code></td>
<td>DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], dtype='datetime64[ns]', freq='2D')</td>
</tr>
</tbody>
</table>
Providing a format argument

In addition to the required datetime string, a `format` argument can be passed to ensure specific parsing. This could also potentially speed up the conversion considerably.

```python
In [51]: pd.to_datetime("2010/11/12", format="%Y/%m/%d")
Out[51]: Timestamp('2010-11-12 00:00:00')

In [52]: pd.to_datetime("12-11-2010 00:00", format="%d-%m-%Y %H:%M")
Out[52]: Timestamp('2010-11-12 00:00:00')
```

For more information on the choices available when specifying the `format` option, see the Python datetime documentation.

Assembling datetime from multiple DataFrame columns

You can also pass a DataFrame of integer or string columns to assemble into a Series of Timestamps.

```python
In [53]: df = pd.DataFrame(
       ....:     {"year": [2015, 2016], "month": [2, 3], "day": [4, 5], "hour": [2, 3]})
       ....:
       ....:

In [54]: pd.to_datetime(df)
Out[54]:
0 2015-02-04 02:00:00
1 2016-03-05 03:00:00
dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```python
In [55]: pd.to_datetime(df["year", "month", "day")
Out[55]:
0 2015-02-04
1 2016-03-05
dtype: datetime64[ns]
```

`pd.to_datetime` looks for standard designations of the datetime component in the column names, including:

- required: year, month, day
- optional: hour, minute, second, millisecond, microsecond, nanosecond

Invalid data

The default behavior, `errors='raise'`, is to raise when unparsable:

```python
In [2]: pd.to_datetime(["2009/07/31", 'asd'], errors='raise')
ValueError: Unknown string format
```

Pass `errors='ignore'` to return the original input when unparsable:

```python
In [56]: pd.to_datetime(["2009/07/31", 'asd'], errors="ignore")
Out[56]: Index(['2009/07/31', 'asd'], dtype='object')
```
Pass `errors='coerce'` to convert unparsable data to NaN (not a time):

```python
In [57]: pd.to_datetime(['2009/07/31', 'asd'], errors='coerce')
Out[57]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

### Epoch timestamps

pandas supports converting integer or float epoch times to `Timestamp` and `DatetimeIndex`. The default unit is nanoseconds, since that is how `Timestamp` objects are stored internally. However, epochs are often stored in another unit which can be specified. These are computed from the starting point specified by the `origin` parameter.

```python
In [58]: pd.to_datetime([1349720105, 1349720105, 1349720105, 1349720105, 1349720105], unit='s')
Out[58]: DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05', '2012-10-10 18:15:05', '2012-10-11 18:15:05', '2012-10-12 18:15:05'], dtype='datetime64[ns]', freq=None)
```

```python
In [59]: pd.to_datetime([1349720105100, 1349720105200, 1349720105300, 1349720105400, 1349720105500], unit='ms')
Out[59]: DatetimeIndex(['2012-10-08 18:15:05.100000', '2012-10-08 18:15:05.200000', '2012-10-08 18:15:05.300000', '2012-10-08 18:15:05.400000', '2012-10-08 18:15:05.500000'], dtype='datetime64[ns]', freq=None)
```

**Note:** The `unit` parameter does not use the same strings as the `format` parameter that was discussed above. The available units are listed on the documentation for `pandas.to_datetime()`.

Changed in version 1.0.0.

Constructing a `Timestamp` or `DatetimeIndex` with an epoch timestamp with the `tz` argument specified will raise a `ValueError`. If you have epochs in wall time in another timezone, you can read the epochs as timezone-naive timestamps and then localize to the appropriate timezone:

```python
In [60]: pd.Timestamp(1262347200000000000).tz_localize('US/Pacific')
Out[60]: Timestamp('2010-01-01 12:00:00-0800', tz='US/Pacific')
```

```python
In [61]: pd.DatetimeIndex([1262347200000000000]).tz_localize('US/Pacific')
Out[61]: DatetimeIndex(['2010-01-01 12:00:00-08:00'], dtype='datetime64[ns, US/Pacific]', freq=None)
```

**Note:** Epoch times will be rounded to the nearest nanosecond.
Warning: Conversion of float epoch times can lead to inaccurate and unexpected results. Python floats have about 15 digits precision in decimal. Rounding during conversion from float to high precision Timestamp is unavoidable. The only way to achieve exact precision is to use a fixed-width types (e.g. an int64).

```python
In [62]: pd.to_datetime([1490195805.433, 1490195805.433502912], unit="s")
Out[62]: DatetimeIndex(['2017-03-22 15:16:45.433000088', '2017-03-22 15:16:45.433502913'], dtype='datetime64[ns]', freq=None)
```

```python
In [63]: pd.to_datetime(1490195805433502912, unit="ns")
Out[63]: Timestamp('2017-03-22 15:16:45.433502912')
```

See also:

Using the origin Parameter

From timestamps to epoch

To invert the operation from above, namely, to convert from a Timestamp to a ‘unix’ epoch:

```python
In [64]: stamps = pd.date_range("2012-10-08 18:15:05", periods=4, freq="D")
In [65]: stamps
Out[65]: DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05', '2012-10-10 18:15:05', '2012-10-11 18:15:05'], dtype='datetime64[ns]', freq='D')
```

We subtract the epoch (midnight at January 1, 1970 UTC) and then floor divide by the “unit” (1 second).

```python
In [66]: (stamps - pd.Timestamp("1970-01-01")) // pd.Timedelta("1s")
Out[66]: Int64Index([1349720105, 1349806505, 1349892905, 1349979305], dtype='int64')
```

Using the origin Parameter

Using the origin parameter, one can specify an alternative starting point for creation of a DatetimeIndex. For example, to use 1960-01-01 as the starting date:

```python
In [67]: pd.to_datetime([1, 2, 3], unit="D", origin=pd.Timestamp("1960-01-01"))
Out[67]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype='datetime64[ns]', freq=None)
```

The default is set at origin='unix', which defaults to 1970-01-01 00:00:00. Commonly called ‘unix epoch’ or POSIX time.

```python
In [68]: pd.to_datetime([1, 2, 3], unit="D")
Out[68]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype='datetime64[ns]', freq=None)
```
2.20.4 Generating ranges of timestamps

To generate an index with timestamps, you can use either the `DatetimeIndex` or `Index` constructor and pass in a list of datetime objects:

```
In [69]: dates = [
       ....:   datetime.datetime(2012, 5, 1),
       ....:   datetime.datetime(2012, 5, 2),
       ....:   datetime.datetime(2012, 5, 3),
       ....: ]
       ....:
# Note the frequency information
In [70]: index = pd.DatetimeIndex(dates)
```

```
In [70]: index
Out[70]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)
```

```
In [71]: index = pd.Index(dates)
```

```
In [72]: index
Out[72]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)
```

In practice this becomes very cumbersome because we often need a very long index with a large number of timestamps. If we need timestamps on a regular frequency, we can use the `date_range()` and `bdate_range()` functions to create a `DatetimeIndex`. The default frequency for `date_range` is a calendar day while the default for `bdate_range` is a business day:

```
In [74]: start = datetime.datetime(2011, 1, 1)
```

```
In [75]: end = datetime.datetime(2012, 1, 1)
```

```
In [76]: index = pd.date_range(start, end)
```

```
In [77]: index
       '2011-01-09', '2011-01-10',
        ...'
       '2011-12-23', '2011-12-24', '2011-12-25', '2011-12-26',
       '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30',
       '2011-12-31', '2012-01-01'],
       dtype='datetime64[ns]', length=366, freq='D')
```

```
In [78]: index = pd.bdate_range(start, end)
```

```
In [79]: index
```

(continues on next page)
Convenience functions like `date_range` and `bdate_range` can utilize a variety of frequency aliases:

```python
In [80]: pd.date_range(start, periods=1000, freq="M")
Out[80]:
               '2011-09-30', '2011-10-31',
               ... '2093-07-31', '2093-08-31', '2093-09-30', '2093-10-31',
               '2093-11-30', '2093-12-31', '2094-01-31', '2094-02-28',
               '2094-03-31', '2094-04-30'],
   dtype='datetime64[ns]', length=1000, freq='M')
```

```python
In [81]: pd.bdate_range(start, periods=250, freq="BQS")
Out[81]:
               '2012-01-02', '2012-04-02', '2012-07-02', '2012-10-01',
               '2013-01-01', '2013-04-01',
               ... '2071-01-01', '2071-04-01', '2071-07-01', '2071-10-01',
               '2072-01-01', '2072-04-01', '2072-07-01', '2072-10-03',
               '2073-01-02', '2073-04-03'],
   dtype='datetime64[ns]', length=250, freq='BQS-JAN')
```

date_range and bdate_range make it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`. The start and end dates are strictly inclusive, so dates outside of those specified will not be generated:

```python
In [82]: pd.date_range(start, end, freq="BM")
Out[82]:
   dtype='datetime64[ns]', freq='BM')
```

```python
In [83]: pd.date_range(start, end, freq="W")
Out[83]:
```

(continues on next page)
Specifying `start`, `end`, and `periods` will generate a range of evenly spaced dates from `start` to `end` inclusively, with `periods` number of elements in the resulting `DatetimeIndex`:

```python
In [86]: pd.date_range("2018-01-01", "2018-01-05", periods=5)
Out[86]:
               '2018-01-05'],
            dtype='datetime64[ns]', freq=None)
```

```python
In [87]: pd.date_range("2018-01-01", "2018-01-05", periods=10)
Out[87]:
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 10:40:00',
               '2018-01-02 18:40:00', '2018-01-03 05:20:00',
               '2018-01-03 16:00:00', '2018-01-04 02:40:00',
               '2018-01-04 13:20:00', '2018-01-05 00:00:00'],
            dtype='datetime64[ns]', freq=None)
```
Custom frequency ranges

`bdate_range` can also generate a range of custom frequency dates by using the `weekmask` and `holidays` parameters. These parameters will only be used if a custom frequency string is passed.

```
In [88]: weekmask = "Mon Wed Fri"

In [89]: holidays = [datetime.datetime(2011, 1, 5), datetime.datetime(2011, 3, 14)]

In [90]: pd.bdate_range(start, end, freq="C", weekmask=weekmask, holidays=holidays)
Out[90]:
               '2011-01-24', '2011-01-26',
               '2011-02-07', '2011-02-10', '2011-02-12', '2011-02-15',
               '2011-02-17', '2011-02-20', '2011-02-22', '2011-02-25',
               '2011-02-27', '2011-02-29', '2011-03-01', '2011-03-03',
               '2011-03-05', '2011-03-08', '2011-03-10', '2011-03-12',
               '2011-08-05', '2011-08-08', '2011-08-10', '2011-08-12',
               '2011-08-15', '2011-08-17', '2011-08-20', '2011-08-22',
               '2011-09-03', '2011-09-05', '2011-09-08', '2011-09-10',
               '2011-10-01', '2011-10-03', '2011-10-05', '2011-10-08',
               '2011-11-27', '2011-11-29', '2011-12-01', '2011-12-03',
               '2011-12-05', '2011-12-08', '2011-12-10', '2011-12-12',
               '2011-12-15', '2011-12-17', '2011-12-19', '2011-12-21',
               '2011-12-24', '2011-12-26', '2011-12-28', '2011-12-30'],
            dtype='datetime64[ns]', length=154, freq='C')

In [91]: pd.bdate_range(start, end, freq="CBMS", weekmask=weekmask)
Out[91]:
DatetimeIndex(['2011-01-03', '2011-02-02', '2011-03-02', '2011-04-01',
               '2011-09-02', '2011-10-02', '2011-11-02'],
            dtype='datetime64[ns]', freq='CBMS')
```

See also:

*Custom business days*

2.20.5 Timestamp limitations

Since pandas represents timestamps in nanosecond resolution, the time span that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [92]: pd.Timestamp.min
Out[92]: Timestamp('1677-09-21 00:12:43.145224193')

In [93]: pd.Timestamp.max
Out[93]: Timestamp('2262-04-11 23:47:16.854775807')
```

See also:

*Representing out-of-bounds spans*
2.20.6 Indexing

One of the main uses for DatetimeIndex is as an index for pandas objects. The DatetimeIndex class contains many time series related optimizations:

- A large range of dates for various offsets are pre-computed and cached under the hood in order to make generating subsequent date ranges very fast (just have to grab a slice).
- Fast shifting using the shift method on pandas objects.
- Unioning of overlapping DatetimeIndex objects with the same frequency is very fast (important for fast data alignment).
- Quick access to date fields via properties such as year, month, etc.
- Regularization functions like snap and very fast asof logic.

DatetimeIndex objects have all the basic functionality of regular Index objects, and a smorgasbord of advanced time series specific methods for easy frequency processing.

See also:

Reindexing methods

Note: While pandas does not force you to have a sorted date index, some of these methods may have unexpected or incorrect behavior if the dates are unsorted.

DatetimeIndex can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```python
In [94]: rng = pd.date_range(start, end, freq="BM")
In [95]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [96]: ts.index
Out[96]:
              dtype='datetime64[ns]', freq='BM')

In [97]: ts[:5].index
Out[97]:
               '2011-05-31'],
              dtype='datetime64[ns]', freq='BM')

In [98]: ts[::2].index
Out[98]:
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```
Partial string indexing

Dates and strings that parse to timestamps can be passed as indexing parameters:

```
In [99]: ts["1/31/2011"]
Out[99]: 0.11920871129693428

In [100]: ts[datetime.datetime(2011, 12, 25):]
Out[100]:
2011-12-30  0.56702
Freq: BM, dtype: float64

In [101]: ts["10/31/2011":"12/31/2011"]
Out[101]:
2011-10-31  0.271860
2011-11-30 -0.424972
2011-12-30  0.567020
Freq: BM, dtype: float64
```

To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

```
In [102]: ts["2011"]
Out[102]:
2011-01-31  0.119209
2011-02-28 -1.044236
2011-03-31 -0.861849
2011-04-29 -2.104569
2011-05-31 -0.494929
2011-06-30  1.071804
2011-07-29  0.721555
2011-08-31 -0.706771
2011-09-30 -1.039575
2011-10-31  0.271860
2011-11-30 -0.424972
2011-12-30  0.567020
Freq: BM, dtype: float64

In [103]: ts["2011-6"]
Out[103]:
2011-06-30  1.071804
Freq: BM, dtype: float64
```

This type of slicing will work on a DataFrame with a DatetimeIndex as well. Since the partial string selection is a form of label slicing, the endpoints will be included. This would include matching times on an included date:

```
Warning: Indexing DataFrame rows with a single string with getitem (e.g. frame[dtstring]) is deprecated starting with pandas 1.2.0 (given the ambiguity whether it is indexing the rows or selecting a column) and will be removed in a future version. The equivalent with .loc (e.g. frame.loc[dtstring]) is still supported.
```

```
In [104]: dft = pd.DataFrame(
        .....: np.random.randn(100000, 1),
        .....: columns=["A"],
```

(continues on next page)
index = pd.date_range("20130101", periods=100000, freq="T"),
.....:

In [105]: dft
Out[105]:
A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00  -1.087401
2013-01-01 00:02:00  -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00  -1.478427
   ...
2013-03-11 10:35:00  -0.747967
2013-03-11 10:36:00  -0.034523
2013-03-11 10:37:00  -0.201754
2013-03-11 10:38:00  -1.509067
2013-03-11 10:39:00  -1.693043
[100000 rows x 1 columns]

In [106]: dft.loc["2013"]
Out[106]:
A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00  -1.087401
2013-01-01 00:02:00  -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00  -1.478427
   ...
2013-03-11 10:35:00  -0.747967
2013-03-11 10:36:00  -0.034523
2013-03-11 10:37:00  -0.201754
2013-03-11 10:38:00  -1.509067
2013-03-11 10:39:00  -1.693043
[100000 rows x 1 columns]

This starts on the very first time in the month, and includes the last date and time for the month:

In [107]: dft["2013-1":"2013-2"]
Out[107]:
A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00  -1.087401
2013-01-01 00:02:00  -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00  -1.478427
   ...
2013-02-28 23:55:00  0.850929
2013-02-28 23:56:00  0.976712
2013-02-28 23:57:00  -2.693884
(continues on next page)
This specifies a stop time that includes all of the times on the last day:

```python
In [108]: dft["2013-1":"2013-2-28"]
Out[108]:
   A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00  1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
... ...
2013-02-28 23:55:00  0.850929
2013-02-28 23:56:00  0.976712
2013-02-28 23:57:00 -2.693884
2013-02-28 23:58:00 -1.575535
2013-02-28 23:59:00 -1.573517
[84960 rows x 1 columns]
```

This specifies an exact stop time (and is not the same as the above):

```python
In [109]: dft["2013-1":"2013-2-28 00:00:00"]
Out[109]:
   A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00  1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
... ...
2013-02-27 23:56:00  0.720521
2013-02-27 23:57:00 -0.072718
2013-02-27 23:58:00 -0.681192
2013-02-27 23:59:00 -0.557501
2013-02-28 00:00:00 -0.557501
[83521 rows x 1 columns]
```

We are stopping on the included end-point as it is part of the index:

```python
In [110]: dft["2013-1-15":"2013-1-15 12:30:00"]
Out[110]:
   A
2013-01-15 00:00:00 -0.984810
2013-01-15 00:01:00  0.941451
2013-01-15 00:02:00  1.559365
2013-01-15 00:03:00  1.034374
2013-01-15 00:04:00 -1.480656
(continues on next page)
```
... ...  
2013-01-15 12:26:00  0.371454  
2013-01-15 12:27:00  -0.930806  
2013-01-15 12:28:00  -0.069177  
2013-01-15 12:29:00  0.066510  
2013-01-15 12:30:00  -0.003945  
[751 rows x 1 columns]  

DatetimeIndex partial string indexing also works on a DataFrame with a MultiIndex:  

In [111]: dft2 = pd.DataFrame(  
      ......: np.random.randn(20, 1),  
      ......: columns=["A"],  
      ......: index=pd.MultiIndex.from_product(  
      ......:     [pd.date_range("20130101", periods=10, freq="12H"), ["a", "b"]  
      ......: ),  
      ......:  
      ......:  
      ......: )  
      ......:  
      
In [112]: dft2  
Out[112]:  

        A  
2013-01-01 00:00:00 a  -0.298694  
   b      0.823553  
2013-01-01 12:00:00 a   0.943285  
   b     -1.479399  
2013-01-02 00:00:00 a   -1.643342  
   b      0.859329  
2013-01-02 12:00:00 a     1.849347  
   b     -1.499416  
2013-01-04 00:00:00 a    0.069036  
   b      0.853616  
2013-01-05 00:00:00 a   0.122297  
   b     1.422060  
2013-01-05 12:00:00 a  0.370079  
   b      1.016331  
   [20 rows x 1 columns]  

In [113]: dft2.loc["2013-01-05"]  
Out[113]:  

        A  
2013-01-05 00:00:00 a   0.122297  
   b     1.422060  
2013-01-05 12:00:00 a   0.370079  
   b     1.016331  

In [114]: idx = pd.IndexSlice  

In [115]: dft2 = dft2.swaplevel(0, 1).sort_index()  

In [116]: dft2.loc[idx[:, "2013-01-05"], :]  
Out[116]:  

        A  
... ...  

New in version 0.25.0.

Slicing with string indexing also honors UTC offset.

```
In [117]: df = pd.DataFrame([0], index=pd.DatetimeIndex(["2019-01-01"], tz="US/Pacific"))
In [118]: df
Out[118]:
0    2019-01-01 00:00:00-08:00 0
In [119]: df["2019-01-01 12:00:00+04:00":"2019-01-01 13:00:00+04:00"]
Out[119]:
0    2019-01-01 00:00:00-08:00 0
```

### Slice vs. exact match

The same string used as an indexing parameter can be treated either as a slice or as an exact match depending on the resolution of the index. If the string is less accurate than the index, it will be treated as a slice, otherwise as an exact match.

Consider a `Series` object with a minute resolution index:

```
In [120]: series_minute = pd.Series([1, 2, 3],
    ...: pd.DatetimeIndex(["2011-12-31 23:59:00", "2012-01-01 00:00:00", "2012-01-01 00:02:00"],
    ...: ))
In [121]: series_minute.index.resolution
Out[121]: 'minute'
```

A timestamp string less accurate than a minute gives a `Series` object.

```
In [122]: series_minute["2011-12-31 23"]
Out[122]:
2011-12-31 23:59:00  1
```

A timestamp string with minute resolution (or more accurate), gives a scalar instead, i.e. it is not casted to a slice.

```
In [123]: series_minute["2011-12-31 23:59"]
Out[123]: 1
```
In [124]: series_minute["2011-12-31 23:59:00"]
Out[124]: 1

If index resolution is second, then the minute-accurate timestamp gives a Series.

In [125]: series_second = pd.Series(
        ....: [1, 2, 3],
        ....: pd.DatetimeIndex(
        ....: ["2011-12-31 23:59:59", "2012-01-01 00:00:00", "2012-01-01 00:00:01"]
        ....: ),
        ....: )

In [126]: series_second.index.resolution
Out[126]: 'second'

In [127]: series_second["2011-12-31 23:59:59"]
Out[127]:
2011-12-31 23:59:59 1
dtype: int64

If the timestamp string is treated as a slice, it can be used to index DataFrame with .loc[] as well.

In [128]: dft_minute = pd.DataFrame(
        ....: {"a": [1, 2, 3], "b": [4, 5, 6]}, index=series_minute.index
        ....: )

In [129]: dft_minute.loc["2011-12-31 23:59:00"]
Out[129]:
a 1
b 4
Name: 2011-12-31 23:59:00, dtype: int64

Warning: However, if the string is treated as an exact match, the selection in DataFrame’s [] will be column-wise and not row-wise, see Indexing Basics. For example dft_minute['2011-12-31 23:59'] will raise KeyError as '2012-12-31 23:59' has the same resolution as the index and there is no column with such name:

To always have unambiguous selection, whether the row is treated as a slice or a single selection, use .loc.

In [130]: dft_minute.loc["2011-12-31 23:59:00"]
Out[130]:
a 1
b 4
Name: 2011-12-31 23:59:00, dtype: int64

Note also that DatetimeIndex resolution cannot be less precise than day.

In [131]: series_monthly = pd.Series(
        ....: [1, 2, 3], pd.DatetimeIndex(["2011-12", "2012-01", "2012-02"])
        ....: )

(continues on next page)
Exact indexing

As discussed in previous section, indexing a `DatetimeIndex` with a partial string depends on the “accuracy” of the period, in other words how specific the interval is in relation to the resolution of the index. In contrast, indexing with `Timestamp` or `datetime` objects is exact, because the objects have exact meaning. These also follow the semantics of including both endpoints.

These `Timestamp` and `datetime` objects have exact hours, minutes, and seconds, even though they were not explicitly specified (they are 0).

```
In [134]: dft[datetime.datetime(2013, 1, 1): datetime.datetime(2013, 2, 28)]
Out[134]:
          A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
   ...
2013-02-27 23:56:00  1.197749
2013-02-27 23:57:00  0.720521
2013-02-27 23:58:00 -0.072718
2013-02-27 23:59:00 -0.681192
2013-02-28 00:00:00 -0.557501
[83521 rows x 1 columns]
```

With no defaults.

```
In [135]: dft[
    ...:    datetime.datetime(2013, 1, 1, 10, 12, 0): datetime.datetime(2013, 2, 28, 10, 12, 0)
    ...:]
Out[135]:
          A
2013-01-01 10:12:00  0.565375
2013-01-01 10:13:00  0.068184
2013-01-01 10:14:00  0.788871
2013-01-01 10:15:00 -0.280343
2013-01-01 10:16:00  0.931536
   ...
```
Truncating & fancy indexing

A `truncate()` convenience function is provided that is similar to slicing. Note that `truncate` assumes a 0 value for any unspecified date component in a `DatetimeIndex` in contrast to slicing which returns any partially matching dates:

```python
In [136]: rng2 = pd.date_range("2011-01-01", "2012-01-01", freq="W")

In [137]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)

In [138]: ts2.truncate(before="2011-11", after="2011-12")
Out[138]:
2011-11-06   0.437823
2011-11-13  -0.293083
2011-11-20  -0.059881
2011-11-27   1.252450
Freq: W-SUN, dtype: float64

In [139]: ts2["2011-11":"2011-12"]
Out[139]:
2011-11-06   0.437823
2011-11-13  -0.293083
2011-11-20  -0.059881
2011-11-27   1.252450
2011-12-04   0.046611
2011-12-11   0.059478
2011-12-18  -0.286539
2011-12-25   0.841669
Freq: W-SUN, dtype: float64
```

Even complicated fancy indexing that breaks the `DatetimeIndex` frequency regularity will result in a `DatetimeIndex`, although frequency is lost:

```python
In [140]: ts2[[0, 2, 6]].index
Out[140]: DatetimeIndex(['2011-01-02', '2011-01-16', '2011-02-13'], dtype='datetime64[ns]', freq=None)
```
2.20.7 Time/date components

There are several time/date properties that one can access from Timestamp or a collection of timestamps like a DatetimeIndex.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>The year of the datetime</td>
</tr>
<tr>
<td>month</td>
<td>The month of the datetime</td>
</tr>
<tr>
<td>day</td>
<td>The days of the datetime</td>
</tr>
<tr>
<td>hour</td>
<td>The hour of the datetime</td>
</tr>
<tr>
<td>minute</td>
<td>The minutes of the datetime</td>
</tr>
<tr>
<td>second</td>
<td>The seconds of the datetime</td>
</tr>
<tr>
<td>microsecond</td>
<td>The microseconds of the datetime</td>
</tr>
<tr>
<td>nanosecond</td>
<td>The nanoseconds of the datetime</td>
</tr>
<tr>
<td>date</td>
<td>Returns datetime.date (does not contain timezone information)</td>
</tr>
<tr>
<td>time</td>
<td>Returns datetime.time (does not contain timezone information)</td>
</tr>
<tr>
<td>timetz</td>
<td>Returns datetime.time as local time with timezone information</td>
</tr>
<tr>
<td>dayofyear</td>
<td>The ordinal day of year</td>
</tr>
<tr>
<td>day_of_year</td>
<td>The ordinal day of year</td>
</tr>
<tr>
<td>weekofyear</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>week</td>
<td>The week ordinal of the year</td>
</tr>
<tr>
<td>dayofweek</td>
<td>The number of the day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>day_of_week</td>
<td>The number of the day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>weekday</td>
<td>The number of the day of the week with Monday=0, Sunday=6</td>
</tr>
<tr>
<td>quarter</td>
<td>Quarter of the date: Jan-Mar = 1, Apr-Jun = 2, etc.</td>
</tr>
<tr>
<td>days_in_month</td>
<td>The number of days in the month of the datetime</td>
</tr>
<tr>
<td>is_month_start</td>
<td>Logical indicating if first day of month (defined by frequency)</td>
</tr>
<tr>
<td>is_month_end</td>
<td>Logical indicating if last day of month (defined by frequency)</td>
</tr>
<tr>
<td>is_quarter_start</td>
<td>Logical indicating if first day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>is_quarter_end</td>
<td>Logical indicating if last day of quarter (defined by frequency)</td>
</tr>
<tr>
<td>is_year_start</td>
<td>Logical indicating if first day of year (defined by frequency)</td>
</tr>
<tr>
<td>is_year_end</td>
<td>Logical indicating if last day of year (defined by frequency)</td>
</tr>
<tr>
<td>is_leap_year</td>
<td>Logical indicating if the date belongs to a leap year</td>
</tr>
</tbody>
</table>

Furthermore, if you have a Series with datetimelike values, then you can access these properties via the `.dt` accessor, as detailed in the section on `.dt accessors`.

New in version 1.1.0.

You may obtain the year, week and day components of the ISO year from the ISO 8601 standard:

```
In [141]: idx = pd.date_range(start="2019-12-29", freq="D", periods=4)

In [142]: idx.isocalendar()
Out[142]:
   year  week  day
2019-12-29  2019   52   7
2019-12-30  2020    1  1
2019-12-31  2020    1  2
2020-01-01  2020    1  3
```

(continues on next page)
In the preceding examples, frequency strings (e.g. 'D') were used to specify a frequency that defined:

- how the date times in DatetimeIndex were spaced when using date_range()
- the frequency of a Period or PeriodIndex

These frequency strings map to a DateOffset object and its subclasses. A DateOffset is similar to a Timedelta that represents a duration of time but follows specific calendar duration rules. For example, a Timedelta day will always increment datetimes by 24 hours, while a DateOffset day will increment datetimes to the same time the next day whether a day represents 23, 24 or 25 hours due to daylight savings time. However, all DateOffset subclasses that are an hour or smaller (Hour, Minute, Second, Milli, Micro, Nano) behave like Timedelta and respect absolute time.

The basic DateOffset acts similar to dateutil.relativedelta (relativedelta documentation) that shifts a date time by the corresponding calendar duration specified. The arithmetic operator (+) can be used to perform the shift.

```
In [144]: ts = pd.Timestamp("2016-10-30 00:00:00", tz="Europe/Helsinki")

# Respects absolute time
In [145]: ts + pd.Timedelta(days=1)
Out[145]: Timestamp('2016-10-30 23:00:00+0200', tz='Europe/Helsinki')

# Respects calendar time
In [146]: ts + pd.DateOffset(days=1)
Out[146]: Timestamp('2016-10-31 00:00:00+0200', tz='Europe/Helsinki')

In [147]: friday = pd.Timestamp("2018-01-05")

In [148]: friday.day_name()
Out[148]: 'Friday'

# Add 2 business days (Friday --> Tuesday)
In [149]: two_business_days = 2 * pd.offsets.BDay()

In [150]: friday + two_business_days
Out[150]: Timestamp('2018-01-09 00:00:00')

In [151]: (friday + two_business_days).day_name()
Out[151]: 'Tuesday'
```

Most DateOffsets have associated frequencies strings, or offset aliases, that can be passed into freq keyword arguments. The available date offsets and associated frequency strings can be found below:
**Date Offsets**

- **DateOffset**
  - None: Generic offset class, defaults to absolute 24 hours
- **BDay** or **BusinessDay**
  - 'B': business day (weekday)
- **CDay** or **CustomBusinessDay**
  - 'C': custom business day
- **Week**
  - 'W': one week, optionally anchored on a day of the week
- **WeekOfMonth**
  - 'WOM': the x-th day of the y-th week of each month
- **LastWeekOfMonth**
  - 'LWOM': the x-th day of the last week of each month
- **MonthEnd**
  - 'M': calendar month end
- **MonthBegin**
  - 'MS': calendar month begin
- **BMonthEnd** or **BusinessMonthEnd**
  - 'BM': business month end
- **BMonthBegin** or **BusinessMonthBegin**
  - 'BMS': business month begin
- **CBMonthEnd** or **CustomBusinessMonthEnd**
  - 'CBM': custom business month end
- **CBMonthBegin** or **CustomBusinessMonthBegin**
  - 'CBMS': custom business month begin
- **SemiMonthEnd**
  - 'SM': 15th (or other day_of_month) and calendar month end
- **SemiMonthBegin**
  - 'SMS': 15th (or other day_of_month) and calendar month begin
- **QuarterEnd**
  - 'Q': calendar quarter end
- **QuarterBegin**
  - 'QS': calendar quarter begin
- **BQuarterEnd**
  - 'BQ': business quarter end
- **BQuarterBegin**
  - 'BQS': business quarter begin
- **FY5253Quarter**
  - 'REQ': retail (aka 52-53 week) quarter
- **YearEnd**
  - 'A': calendar year end
- **YearBegin**
  - 'AS' or 'BYS': calendar year begin
- **BYearEnd**
  - 'BA': business year end
- **BYearBegin**
  - 'BAS': business year begin
- **FY5253**
  - 'RE': retail (aka 52-53 week) year
- **Easter**
  - None: Easter holiday
- **BusinessHour**
  - 'BH': business hour
- **CustomBusinessHour**
  - 'CBH': custom business hour
- **Day**
  - 'D': one absolute day
- **Hour**
  - 'H': one hour
- **Minute**
  - 'T' or 'min': one minute
- **Second**
  - 'S': one second
- **Milli**
  - 'L' or 'ms': one millisecond
- **Micro**
  - 'U' or 'us': one microsecond
- **Nano**
  - 'N': one nanosecond

**DateOffsets** additionally have `rollforward()` and `rollback()` methods for moving a date forward or backward respectively to a valid offset date relative to the offset. For example, business offsets will roll dates that land on the weekends (Saturday and Sunday) forward to Monday since business offsets operate on the weekdays.

```
In [152]: ts = pd.Timestamp("2018-01-06 00:00:00")
```
In [153]: ts.day_name()
Out[153]: 'Saturday'

# BusinessHour's valid offset dates are Monday through Friday
In [154]: offset = pd.offsets.BusinessHour(start="09:00")

# Bring the date to the closest offset date (Monday)
In [155]: offset.rollforward(ts)
Out[155]: Timestamp('2018-01-08 09:00:00')

# Date is brought to the closest offset date first and then the hour is added
In [156]: ts + offset
Out[156]: Timestamp('2018-01-08 10:00:00')

These operations preserve time (hour, minute, etc) information by default. To reset time to midnight, use normalize() before or after applying the operation (depending on whether you want the time information included in the operation).

In [157]: ts = pd.Timestamp("2014-01-01 09:00")
In [158]: day = pd.offsets.Day()
In [159]: day + ts
Out[159]: Timestamp('2014-01-02 09:00:00')

In [160]: (day + ts).normalize()
Out[160]: Timestamp('2014-01-02 00:00:00')

In [161]: ts = pd.Timestamp("2014-01-01 22:00")
In [162]: hour = pd.offsets.Hour()
In [163]: hour + ts
Out[163]: Timestamp('2014-01-01 23:00:00')

In [164]: (hour + ts).normalize()
Out[164]: Timestamp('2014-01-01 00:00:00')

In [165]: (hour + pd.Timestamp("2014-01-01 23:30")).normalize()
Out[165]: Timestamp('2014-01-02 00:00:00')

**Parametric offsets**

Some of the offsets can be “parameterized” when created to result in different behaviors. For example, the Week offset for generating weekly data accepts a weekday parameter which results in the generated dates always lying on a particular day of the week:

In [166]: d = datetime.datetime(2008, 8, 18, 9, 0)
In [167]: d
Out[167]: datetime.datetime(2008, 8, 18, 9, 0)

(continues on next page)
In [168]: d + pd.offsets.Week()
Out[168]: Timestamp('2008-08-25 09:00:00')

In [169]: d + pd.offsets.Week(weekday=4)
Out[169]: Timestamp('2008-08-22 09:00:00')

In [170]: (d + pd.offsets.Week(weekday=4)).weekday()
Out[170]: 4

In [171]: d - pd.offsets.Week()
Out[171]: Timestamp('2008-08-11 09:00:00')

The normalize option will be effective for addition and subtraction.

In [172]: d + pd.offsets.Week(normalize=True)
Out[172]: Timestamp('2008-08-25 00:00:00')

In [173]: d - pd.offsets.Week(normalize=True)
Out[173]: Timestamp('2008-08-11 00:00:00')

Another example is parameterizing YearEnd with the specific ending month:

In [174]: d + pd.offsets.YearEnd()
Out[174]: Timestamp('2008-12-31 09:00:00')

In [175]: d + pd.offsets.YearEnd(month=6)
Out[175]: Timestamp('2009-06-30 09:00:00')

Using offsets with Series / DatetimeIndex

Offsets can be used with either a Series or DatetimeIndex to apply the offset to each element.

In [176]: rng = pd.date_range("2012-01-01", "2012-01-03")

In [177]: s = pd.Series(rng)

In [178]: rng
Out[178]: DatetimeIndex(['2012-01-01', '2012-01-02', '2012-01-03'], dtype='datetime64[ns] →', freq='D')

In [179]: rng + pd.DateOffset(months=2)
Out[179]: DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype='datetime64[ns] →', freq=None)

In [180]: s + pd.DateOffset(months=2)
Out[180]:
0   2012-03-01
1   2012-03-02
2   2012-03-03
dtype: datetime64[ns]

(continues on next page)
If the offset class maps directly to a Timedelta (Day, Hour, Minute, Second, Micro, Milli, Nano) it can be used exactly like a Timedelta - see the Timedelta section for more examples.

```
In [182]: s - pd.offsets.Day(2)
Out[182]:
0  2011-12-30
1  2011-12-31
2  2012-01-01
dtype: datetime64[ns]
```

```
In [183]: td = s - pd.Series(pd.date_range("2011-12-29", "2011-12-31"))
```

```
In [184]: td
Out[184]:
0   3 days
1   3 days
2   3 days
dtype: timedelta64[ns]
```

```
In [185]: td + pd.offsets.Minute(15)
Out[185]:
0  3 days 00:15:00
1  3 days 00:15:00
2  3 days 00:15:00
dtype: timedelta64[ns]
```

Note that some offsets (such as BQuarterEnd) do not have a vectorized implementation. They can still be used but may calculate significantly slower and will show a PerformanceWarning.

```
In [186]: rng + pd.offsets.BQuarterEnd()
Out[186]: DatetimeIndex(['2012-03-30', '2012-03-30', '2012-03-30'],
dtype='datetime64[ns]', freq=None)
```

**Custom business days**

The CDay or CustomBusinessDay class provides a parametric BusinessDay class which can be used to create customized business day calendars which account for local holidays and local weekend conventions.

As an interesting example, let’s look at Egypt where a Friday-Saturday weekend is observed.

```
In [187]: weekmask_egypt = "Sun Mon Tue Wed Thu"
# They also observe International Workers' Day so let's
# add that for a couple of years
In [188]: holidays = [
```
.....:    "2012-05-01",
.....:    datetime.datetime(2013, 5, 1),
.....:    np.datetime64("2014-05-01"),
.....:    ]
.....:

In [189]:
bday_egypt = pd.offsets.CustomBusinessDay(
.....:    holidays=holidays,
.....:    weekmask=weekmask_egypt,
.....:    )
.....:

In [190]:
dt = datetime.datetime(2013, 4, 30)

In [191]:
dt + 2 * bday_egypt
Out[191]:
Timestamp('2013-05-05 00:00:00')

Let's map to the weekday names:

In [192]:
dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [193]:
pd.Series(dts.weekday, dts).map(pd.Series("Mon Tue Wed Thu Fri Sat Sun".split()))
Out[193]:
2013-04-30 Tue
2013-05-02 Thu
2013-05-05 Sun
2013-05-06 Mon
2013-05-07 Tue
Freq: C, dtype: object

Holiday calendars can be used to provide the list of holidays. See the holiday calendar section for more information.

In [194]:
from pandas.tseries.holiday import USFederalHolidayCalendar

In [195]:
bday_us = pd.offsets.CustomBusinessDay(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [196]:
dt = datetime.datetime(2014, 1, 17)

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [197]:
dt + bday_us
Out[197]:
Timestamp('2014-01-21 00:00:00')

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.

In [198]:
bmth_us = pd.offsets.CustomBusinessMonthBegin(calendar=USFederalHolidayCalendar())

# Skip new years
In [199]:
dt = datetime.datetime(2013, 12, 17)

In [200]:
dt + bmth_us

(continues on next page)
Out[200]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [201]: pd.date_range(start="20100101", end="20120101", freq=bmth_us)
Out[201]:
  DatetimeIndex(['2010-01-04', '2010-02-01', '2010-03-01', '2010-04-01',
  '2010-05-03', '2010-06-01', '2010-07-01', '2010-08-02',
  '2010-09-01', '2010-10-01', '2010-11-01', '2010-12-01',
  '2011-01-03', '2011-02-01', '2011-03-01', '2011-04-01',
  '2011-09-01', '2011-10-03', '2011-11-01', '2011-12-01'],
  dtype='datetime64[ns]', freq='CBMS')

Note: The frequency string ‘C’ is used to indicate that a CustomBusinessDay DateOffset is used, it is important to note that since CustomBusinessDay is a parameterised type, instances of CustomBusinessDay may differ and this is not detectable from the ‘C’ frequency string. The user therefore needs to ensure that the ‘C’ frequency string is used consistently within the user’s application.

Business hour

The BusinessHour class provides a business hour representation on BusinessDay, allowing to use specific start and end times.

By default, BusinessHour uses 9:00 - 17:00 as business hours. Adding BusinessHour will increment Timestamp by hourly frequency. If target Timestamp is out of business hours, move to the next business hour then increment it. If the result exceeds the business hours end, the remaining hours are added to the next business day.

In [202]: bh = pd.offsets.BusinessHour()

In [203]: bh
Out[203]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [204]: pd.Timestamp("2014-08-01 10:00").weekday()
Out[204]: 4

In [205]: pd.Timestamp("2014-08-01 10:00") + bh
Out[205]: Timestamp('2014-08-01 11:00:00')

# Below example is the same as: pd.Timestamp('2014-08-01 09:00') + bh
In [206]: pd.Timestamp("2014-08-01 08:00") + bh
Out[206]: Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
In [207]: pd.Timestamp("2014-08-01 16:00") + bh
Out[207]: Timestamp('2014-08-04 09:00:00')

# Remainings are added to the next day
In [208]: pd.Timestamp("2014-08-01 16:30") + bh
Out[208]: Timestamp('2014-08-04 09:30:00')

(continues on next page)
# Adding 2 business hours

In [209]: pd.Timestamp("2014-08-01 10:00") + pd.offsets.BusinessHour(2)
Out[209]: Timestamp('2014-08-01 12:00:00')

# Subtracting 3 business hours

In [210]: pd.Timestamp("2014-08-01 10:00") + pd.offsets.BusinessHour(-3)
Out[210]: Timestamp('2014-07-31 15:00:00')

You can also specify start and end time by keywords. The argument must be a str with an hour:minute representation or a datetime.time instance. Specifying seconds, microseconds and nanoseconds as business hour results in ValueError.

In [211]: bh = pd.offsets.BusinessHour(start="11:00", end=datetime.time(20, 0))

In [212]: bh
Out[212]: <BusinessHour: BH=11:00-20:00>

In [213]: pd.Timestamp("2014-08-01 13:00") + bh
Out[213]: Timestamp('2014-08-01 14:00:00')

In [214]: pd.Timestamp("2014-08-01 09:00") + bh
Out[214]: Timestamp('2014-08-01 12:00:00')

In [215]: pd.Timestamp("2014-08-01 18:00") + bh
Out[215]: Timestamp('2014-08-01 19:00:00')

Passing start time later than end represents midnight business hour. In this case, business hour exceeds midnight and overlap to the next day. Valid business hours are distinguished by whether it started from valid BusinessDay.

In [216]: bh = pd.offsets.BusinessHour(start="17:00", end="09:00")

In [217]: bh
Out[217]: <BusinessHour: BH=17:00-09:00>

In [218]: pd.Timestamp("2014-08-01 17:00") + bh
Out[218]: Timestamp('2014-08-01 18:00:00')

In [219]: pd.Timestamp("2014-08-01 23:00") + bh
Out[219]: Timestamp('2014-08-02 00:00:00')

# Although 2014-08-02 is Saturday,
# it is valid because it starts from 08-01 (Friday).
In [220]: pd.Timestamp("2014-08-02 04:00") + bh
Out[220]: Timestamp('2014-08-02 05:00:00')

# Although 2014-08-04 is Monday,
# it is out of business hours because it starts from 08-03 (Sunday).
In [221]: pd.Timestamp("2014-08-04 04:00") + bh
Out[221]: Timestamp('2014-08-04 18:00:00')

Applying BusinessHour.rollforward and rollback to out of business hours results in the next business hour start or previous day’s end. Different from other offsets, BusinessHour.rollforward may output different results from
apply by definition.

This is because one day’s business hour end is equal to next day’s business hour start. For example, under the default business hours (9:00 - 17:00), there is no gap (0 minutes) between 2014-08-01 17:00 and 2014-08-04 09:00.

```
# This adjusts a Timestamp to business hour edge
In [222]: pd.offsets.BusinessHour().rollback(pd.Timestamp("2014-08-02 15:00"))
Out[222]: Timestamp('2014-08-01 17:00:00')

In [232]: pd.offsets.BusinessHour().rollforward(pd.Timestamp("2014-08-02 15:00"))
Out[232]: Timestamp('2014-08-04 09:00:00')

In [223]: pd.offsets.BusinessHour().rollforward(pd.Timestamp("2014-08-02 15:00"))
Out[223]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessHour() + pd.Timestamp('2014-08-01 17:00').
# And it is the same as BusinessHour() + pd.Timestamp('2014-08-04 09:00')
In [224]: pd.offsets.BusinessHour() + pd.Timestamp("2014-08-02 15:00")
Out[224]: Timestamp('2014-08-04 10:00:00')

# BusinessDay results (for reference)
In [225]: pd.offsets.BusinessHour().rollforward(pd.Timestamp("2014-08-02"))
Out[225]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessDay() + pd.Timestamp('2014-08-01')
# The result is the same as rollworward because BusinessDay never overlap.
In [226]: pd.offsets.BusinessHour() + pd.Timestamp("2014-08-02")
Out[226]: Timestamp('2014-08-04 10:00:00')
```

BusinessHour regards Saturday and Sunday as holidays. To use arbitrary holidays, you can use CustomBusinessHour offset, as explained in the following subsection.

### Custom business hour

The CustomBusinessHour is a mixture of BusinessHour and CustomBusinessDay which allows you to specify arbitrary holidays. CustomBusinessHour works as the same as BusinessHour except that it skips specified custom holidays.

```
In [227]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [228]: bhour_us = pd.offsets.CustomBusinessHour(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [229]: dt = datetime.datetime(2014, 1, 17, 15)

In [230]: dt + bhour_us
Out[230]: Timestamp('2014-01-17 16:00:00')

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [231]: dt + bhour_us * 2
Out[231]: Timestamp('2014-01-21 09:00:00')
```

You can use keyword arguments supported by either BusinessHour and CustomBusinessDay.

```
In [232]: bhour_mon = pd.offsets.CustomBusinessHour(start="10:00", weekmask="\nFri")
```
# Monday is skipped because it's a holiday, business hour starts from 10:00

In [233]: dt + bhour_mon * 2

Out[233]: Timestamp('2014-01-21 10:00:00')

## Offset aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as *offset aliases*.

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>business day frequency</td>
</tr>
<tr>
<td>C</td>
<td>custom business day frequency</td>
</tr>
<tr>
<td>D</td>
<td>calendar day frequency</td>
</tr>
<tr>
<td>W</td>
<td>weekly frequency</td>
</tr>
<tr>
<td>M</td>
<td>month end frequency</td>
</tr>
<tr>
<td>SM</td>
<td>semi-month end frequency (15th and end of month)</td>
</tr>
<tr>
<td>BM</td>
<td>business month end frequency</td>
</tr>
<tr>
<td>CBM</td>
<td>custom business month end frequency</td>
</tr>
<tr>
<td>MS</td>
<td>month start frequency</td>
</tr>
<tr>
<td>SMS</td>
<td>semi-month start frequency (1st and 15th)</td>
</tr>
<tr>
<td>BMS</td>
<td>business month start frequency</td>
</tr>
<tr>
<td>CBMS</td>
<td>custom business month start frequency</td>
</tr>
<tr>
<td>Q</td>
<td>quarter end frequency</td>
</tr>
<tr>
<td>BQ</td>
<td>business quarter end frequency</td>
</tr>
<tr>
<td>QS</td>
<td>quarter start frequency</td>
</tr>
<tr>
<td>BQS</td>
<td>business quarter start frequency</td>
</tr>
<tr>
<td>A, Y</td>
<td>year end frequency</td>
</tr>
<tr>
<td>BA, BY</td>
<td>business year end frequency</td>
</tr>
<tr>
<td>AS, YS</td>
<td>year start frequency</td>
</tr>
<tr>
<td>BAS, BYS</td>
<td>business year start frequency</td>
</tr>
<tr>
<td>BH</td>
<td>business hour frequency</td>
</tr>
<tr>
<td>H</td>
<td>hourly frequency</td>
</tr>
<tr>
<td>T, min</td>
<td>minutely frequency</td>
</tr>
<tr>
<td>S</td>
<td>secondly frequency</td>
</tr>
<tr>
<td>L, ms</td>
<td>millisecond</td>
</tr>
<tr>
<td>U, us</td>
<td>microseconds</td>
</tr>
<tr>
<td>N</td>
<td>nanoseconds</td>
</tr>
</tbody>
</table>

**Note:**

When using the offset aliases above, it should be noted that functions such as *date_range()* and *bdate_range()* will only return timestamps that are in the interval defined by the *start_date* and *end_date*. If the *start_date* does not correspond to the frequency, the returned timestamps will start at the next valid timestamp, same for *end_date*, the returned timestamps will stop at the previous valid timestamp.

For example, for the offset MS, if the *start_date* is not the first of the month, the returned timestamps will start with the first day of the next month. If *end_date* is not the first day of a month, the last returned timestamp will be the first day of the corresponding month.
In [234]: dates_lst_1 = pd.date_range("2020-01-06", "2020-04-03", freq="MS")

In [235]: dates_lst_1
Out[235]: DatetimeIndex(['2020-02-01', '2020-03-01', '2020-04-01'], dtype='datetime64[ns]', freq='MS')

In [236]: dates_lst_2 = pd.date_range("2020-01-01", "2020-04-01", freq="MS")

In [237]: dates_lst_2
Out[237]: DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01', '2020-04-01'], dtype='datetime64[ns]', freq='MS')

We can see in the above example date_range() and bdate_range() will only return the valid timestamps between the start_date and end_date. If these are not valid timestamps for the given frequency it will roll to the next value for start_date (respectively previous for the end_date).

Combining aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

In [238]: pd.date_range(start, periods=5, freq="B")

In [239]: pd.date_range(start, periods=5, freq=pd.offsets.BDay())

You can combine together day and intraday offsets:

In [240]: pd.date_range(start, periods=10, freq="2h20min")
Out[240]: DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 02:20:00', '2011-01-01 04:40:00', '2011-01-01 07:00:00', '2011-01-01 09:20:00', '2011-01-01 11:40:00', '2011-01-01 14:00:00', '2011-01-01 16:20:00', '2011-01-01 18:40:00', '2011-01-01 21:00:00'], dtype='datetime64[ns]', freq='140T')

In [241]: pd.date_range(start, periods=10, freq="1D10U")
Out[241]: DatetimeIndex(['2011-01-01 00:00:00', '2011-01-02 00:00:00.000010', '2011-01-03 00:00:00.000020', '2011-01-04 00:00:00.000030', '2011-01-05 00:00:00.000040', '2011-01-06 00:00:00.000050', '2011-01-07 00:00:00.000060', '2011-01-08 00:00:00.000070', '2011-01-09 00:00:00.000080', '2011-01-10 00:00:00.000090'], dtype='datetime64[ns]', freq='86400000010U')
## Anchored offsets

For some frequencies you can specify an anchoring suffix:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W-SUN</td>
<td>weekly frequency (Sundays). Same as ‘W’</td>
</tr>
<tr>
<td>W-MON</td>
<td>weekly frequency (Mondays)</td>
</tr>
<tr>
<td>W-TUE</td>
<td>weekly frequency (Tuesdays)</td>
</tr>
<tr>
<td>W-WED</td>
<td>weekly frequency (Wednesdays)</td>
</tr>
<tr>
<td>W-THU</td>
<td>weekly frequency (Thursdays)</td>
</tr>
<tr>
<td>W-FRI</td>
<td>weekly frequency (Fridays)</td>
</tr>
<tr>
<td>W-SAT</td>
<td>weekly frequency (Saturdays)</td>
</tr>
<tr>
<td>(B)Q(S)- DEC</td>
<td>quarterly frequency, year ends in December. Same as ‘Q’</td>
</tr>
<tr>
<td>(B)Q(S)- JAN</td>
<td>quarterly frequency, year ends in January</td>
</tr>
<tr>
<td>(B)Q(S)- FEB</td>
<td>quarterly frequency, year ends in February</td>
</tr>
<tr>
<td>(B)Q(S)- MAR</td>
<td>quarterly frequency, year ends in March</td>
</tr>
<tr>
<td>(B)Q(S)- APR</td>
<td>quarterly frequency, year ends in April</td>
</tr>
<tr>
<td>(B)Q(S)- MAY</td>
<td>quarterly frequency, year ends in May</td>
</tr>
<tr>
<td>(B)Q(S)- JUN</td>
<td>quarterly frequency, year ends in June</td>
</tr>
<tr>
<td>(B)Q(S)- JUL</td>
<td>quarterly frequency, year ends in July</td>
</tr>
<tr>
<td>(B)Q(S)- AUG</td>
<td>quarterly frequency, year ends in August</td>
</tr>
<tr>
<td>(B)Q(S)- SEP</td>
<td>quarterly frequency, year ends in September</td>
</tr>
<tr>
<td>(B)Q(S)- OCT</td>
<td>quarterly frequency, year ends in October</td>
</tr>
<tr>
<td>(B)Q(S)- NOV</td>
<td>quarterly frequency, year ends in November</td>
</tr>
<tr>
<td>(B)A(S)- DEC</td>
<td>annual frequency, anchored end of December. Same as ‘A’</td>
</tr>
<tr>
<td>(B)A(S)- JAN</td>
<td>annual frequency, anchored end of January</td>
</tr>
<tr>
<td>(B)A(S)- FEB</td>
<td>annual frequency, anchored end of February</td>
</tr>
<tr>
<td>(B)A(S)- MAR</td>
<td>annual frequency, anchored end of March</td>
</tr>
<tr>
<td>(B)A(S)- APR</td>
<td>annual frequency, anchored end of April</td>
</tr>
<tr>
<td>(B)A(S)- MAY</td>
<td>annual frequency, anchored end of May</td>
</tr>
<tr>
<td>(B)A(S)- JUN</td>
<td>annual frequency, anchored end of June</td>
</tr>
<tr>
<td>(B)A(S)- JUL</td>
<td>annual frequency, anchored end of July</td>
</tr>
</tbody>
</table>

continues on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B)A(S)-AUG</td>
<td>annual frequency, anchored end of August</td>
</tr>
<tr>
<td>(B)A(S)-SEP</td>
<td>annual frequency, anchored end of September</td>
</tr>
<tr>
<td>(B)A(S)-OCT</td>
<td>annual frequency, anchored end of October</td>
</tr>
<tr>
<td>(B)A(S)-NOV</td>
<td>annual frequency, anchored end of November</td>
</tr>
</tbody>
</table>

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

**Anchored offset semantics**

For those offsets that are anchored to the start or end of specific frequency (`MonthEnd`, `MonthBegin`, `WeekEnd`, etc), the following rules apply to rolling forward and backwards.

When \( n \) is not 0, if the given date is not on an anchor point, it snapped to the next(previous) anchor point, and moved \(|n|−1\) additional steps forwards or backwards.

```
In [242]: pd.Timestamp("2014-01-02") + pd.offsets.MonthBegin(n=1)
Out[242]: Timestamp('2014-02-01 00:00:00')

In [243]: pd.Timestamp("2014-01-02") + pd.offsets.MonthEnd(n=1)
Out[243]: Timestamp('2014-01-31 00:00:00')

In [244]: pd.Timestamp("2014-01-01") - pd.offsets.MonthBegin(n=1)
Out[244]: Timestamp('2013-12-01 00:00:00')

In [245]: pd.Timestamp("2014-01-31") - pd.offsets.MonthEnd(n=1)
Out[245]: Timestamp('2013-12-31 00:00:00')
```

In [246]: pd.Timestamp("2014-01-02") + pd.offsets.MonthBegin(n=4)
Out[246]: Timestamp('2014-05-01 00:00:00')

In [247]: pd.Timestamp("2014-01-02") - pd.offsets.MonthBegin(n=4)
Out[247]: Timestamp('2013-10-01 00:00:00')

```
In [248]: pd.Timestamp("2014-01-01") + pd.offsets.MonthBegin(n=1)
Out[248]: Timestamp('2014-02-01 00:00:00')

In [249]: pd.Timestamp("2014-01-31") + pd.offsets.MonthEnd(n=1)
Out[249]: Timestamp('2014-02-28 00:00:00')

In [250]: pd.Timestamp("2014-01-01") - pd.offsets.MonthBegin(n=1)
Out[250]: Timestamp('2013-12-01 00:00:00')

In [251]: pd.Timestamp("2014-01-31") - pd.offsets.MonthEnd(n=1)
Out[251]: Timestamp('2013-12-31 00:00:00')
```

If the given date is on an anchor point, it is moved \(|n|\) points forwards or backwards.

```
In [248]: pd.Timestamp("2014-01-02") + pd.offsets.MonthBegin(n=1)
Out[248]: Timestamp('2014-02-01 00:00:00')

In [249]: pd.Timestamp("2014-01-31") + pd.offsets.MonthEnd(n=1)
Out[249]: Timestamp('2014-02-28 00:00:00')

In [250]: pd.Timestamp("2014-01-01") - pd.offsets.MonthBegin(n=1)
Out[250]: Timestamp('2013-12-01 00:00:00')

In [251]: pd.Timestamp("2014-01-31") - pd.offsets.MonthEnd(n=1)
Out[251]: Timestamp('2013-12-31 00:00:00')
```

(continues on next page)
In [252]: pd.Timestamp("2014-01-01") + pd.offsets.MonthBegin(n=4)
Out[252]: Timestamp('2014-05-01 00:00:00')

In [253]: pd.Timestamp("2014-01-31") - pd.offsets.MonthBegin(n=4)
Out[253]: Timestamp('2013-10-01 00:00:00')

For the case when n=0, the date is not moved if on an anchor point, otherwise it is rolled forward to the next anchor point.

In [254]: pd.Timestamp("2014-01-02") + pd.offsets.MonthBegin(n=0)
Out[254]: Timestamp('2014-02-01 00:00:00')

In [255]: pd.Timestamp("2014-01-02") + pd.offsets.MonthEnd(n=0)
Out[255]: Timestamp('2014-01-31 00:00:00')

In [256]: pd.Timestamp("2014-01-01") + pd.offsets.MonthBegin(n=0)
Out[256]: Timestamp('2014-01-01 00:00:00')

In [257]: pd.Timestamp("2014-01-31") + pd.offsets.MonthEnd(n=0)
Out[257]: Timestamp('2014-01-31 00:00:00')

Holidays / holiday calendars

Holidays and calendars provide a simple way to define holiday rules to be used with CustomBusinessDay or in other analysis that requires a predefined set of holidays. The AbstractHolidayCalendar class provides all the necessary methods to return a list of holidays and only rules need to be defined in a specific holiday calendar class. Furthermore, the start_date and end_date class attributes determine over what date range holidays are generated. These should be overwritten on the AbstractHolidayCalendar class to have the range apply to all calendar subclasses. USFederalHolidayCalendar is the only calendar that exists and primarily serves as an example for developing other calendars.

For holidays that occur on fixed dates (e.g., US Memorial Day or July 4th) an observance rule determines when that holiday is observed if it falls on a weekend or some other non-observed day. Defined observance rules are:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nearest_workday</td>
<td>move Saturday to Friday and Sunday to Monday</td>
</tr>
<tr>
<td>sunday_to_monday</td>
<td>move Sunday to following Monday</td>
</tr>
<tr>
<td>next_monday_or_tuesday</td>
<td>Saturday to Monday and Sunday/Monday to Tuesday</td>
</tr>
<tr>
<td>previous_friday</td>
<td>move Saturday and Sunday to previous Friday</td>
</tr>
<tr>
<td>next_monday</td>
<td>move Saturday and Sunday to following Monday</td>
</tr>
</tbody>
</table>

An example of how holidays and holiday calendars are defined:

In [258]: from pandas.tseries.holiday import (  
        Holiday,  
        USMemorialDay,  
        AbstractHolidayCalendar,  
        nearest_workday,  
        MO,  
        (continues on next page)
In [259]: class ExampleCalendar(AbstractHolidayCalendar):
....:     rules = [
....:         USMemorialDay,
....:         Holiday("July 4th", month=7, day=4, observance=nearest_workday),
....:         Holiday("Columbus Day",
....:              month=10,
....:              day=1,
....:              offset=pd.DateOffset(weekday=MO(2)),
....:         ),
....:     ]

In [260]: cal = ExampleCalendar()

In [261]: cal.holidays(datetime.datetime(2012, 1, 1), datetime.datetime(2012, 12, 31))
Out[261]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype='datetime64[ns]')

hint weekday=MO(2) is same as 2 * Week(weekday=2)

Using this calendar, creating an index or doing offset arithmetic skips weekends and holidays (i.e., Memorial Day/July 4th). For example, the below defines a custom business day offset using the ExampleCalendar. Like any other offset, it can be used to create a DatetimeIndex or added to datetime or Timestamp objects.

In [262]: pd.date_range(
....:     start="7/1/2012", end="7/10/2012", freq=pd.offsets.CDay(calendar=cal)
....: ).to_pydatetime()
Out[262]: array(['2012-07-02', '2012-07-03', '2012-07-04', '2012-07-05', '2012-07-06', '2012-07-09'], dtype=object)

In [263]: offset = pd.offsets.CustomBusinessDay(calendar=cal)

In [264]: datetime.datetime(2012, 5, 25) + offset
Out[264]: Timestamp('2012-05-29 00:00:00')

In [265]: datetime.datetime(2012, 7, 3) + offset
Out[265]: Timestamp('2012-07-05 00:00:00')

In [266]: datetime.datetime(2012, 7, 3) + 2 * offset
Out[266]: Timestamp('2012-07-07 00:00:00')

In [267]: datetime.datetime(2012, 7, 6) + offset
Out[267]: Timestamp('2012-07-09 00:00:00')
Ranges are defined by the `start_date` and `end_date` class attributes of `AbstractHolidayCalendar`. The defaults are shown below.

```python
In [268]: AbstractHolidayCalendar.start_date
Out[268]: Timestamp('1970-01-01 00:00:00')

In [269]: AbstractHolidayCalendar.end_date
Out[269]: Timestamp('2200-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as datetime/Timestamp/string.

```python
In [270]: AbstractHolidayCalendar.start_date = datetime.datetime(2012, 1, 1)
In [271]: AbstractHolidayCalendar.end_date = datetime.datetime(2012, 12, 31)
In [272]: cal.holidays()
Out[272]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype='datetime64[ns]', freq=None)
```

Every calendar class is accessible by name using the `get_calendar` function which returns a holiday class instance. Any imported calendar class will automatically be available by this function. Also, `HolidayCalendarFactory` provides an easy interface to create calendars that are combinations of calendars or calendars with additional rules.

```python
In [273]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory,
   ...:     USLaborDay

In [274]: cal = get_calendar("ExampleCalendar")

In [275]: cal.rules
Out[275]: [Holiday: Memorial Day (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
          Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x7f2ca6999430>),
          Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]

In [276]: new_cal = HolidayCalendarFactory("NewExampleCalendar", cal, USLaborDay)

In [277]: new_cal.rules
Out[277]: [Holiday: Labor Day (month=9, day=1, offset=<DateOffset: weekday=MO(+1)>),
          Holiday: Memorial Day (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
          Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x7f2ca6999430>),
          Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]
```
2.20.9 Time series-related instance methods

Shifting / lagging

One may want to shift or lag the values in a time series back and forward in time. The method for this is `shift()`, which is available on all of the pandas objects.

```python
In [278]: ts = pd.Series(range(len(rng)), index=rng)
In [279]: ts = ts[:5]
In [280]: ts.shift(1)
Out[280]:
2012-01-01    NaN
2012-01-02    0.0
2012-01-03    1.0
Freq: D, dtype: float64
```

The `shift` method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object or also an offset alias.

When `freq` is specified, `shift` method changes all the dates in the index rather than changing the alignment of the data and the index:

```python
In [281]: ts.shift(5, freq="D")
Out[281]:
2012-01-06    0
2012-01-07    1
2012-01-08    2
Freq: D, dtype: int64

In [282]: ts.shift(5, freq=pd.offsets.BDay())
Out[282]:
2012-01-06    0
2012-01-09    1
2012-01-10    2
dtype: int64

In [283]: ts.shift(5, freq="BM")
Out[283]:
2012-05-31    0
2012-05-31    1
2012-05-31    2
dtype: int64
```

Note that with when `freq` is specified, the leading entry is no longer NaN because the data is not being realigned.
Frequency conversion

The primary function for changing frequencies is the `asfreq()` method. For a DatetimeIndex, this is basically just a thin, but convenient wrapper around `reindex()` which generates a `date_range` and calls `reindex`.

```python
In [284]: dr = pd.date_range("1/1/2010", periods=3, freq=3 * pd.offsets.BDay())

In [285]: ts = pd.Series(np.random.randn(3), index=dr)

In [286]: ts
Out[286]:
2010-01-01  1.494522
2010-01-06 -0.778425
2010-01-11 -0.253355
Freq: 3B, dtype: float64

In [287]: ts.asfreq(pd.offsets.BDay())
Out[287]:
2010-01-01  1.494522
2010-01-04  NaN
2010-01-05  NaN
2010-01-06 -0.778425
2010-01-07  NaN
2010-01-08  NaN
2010-01-11 -0.253355
Freq: B, dtype: float64
```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion.

```python
In [288]: ts.asfreq(pd.offsets.BDay(), method="pad")
Out[288]:
2010-01-01  1.494522
2010-01-04  1.494522
2010-01-05  1.494522
2010-01-06 -0.778425
2010-01-07 -0.778425
2010-01-08 -0.778425
2010-01-11 -0.253355
Freq: B, dtype: float64
```
Filling forward / backward

Related to asfreq and reindex is `fillna()`, which is documented in the missing data section.

Converting to Python datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.

2.20.10 Resampling

Pandas has a simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications.

`resample()` is a time-based groupby, followed by a reduction method on each of its groups. See some cookbook examples for some advanced strategies.

The `resample()` method can be used directly from `DataFrameGroupBy` objects, see the groupby docs.

Basics

```python
In [289]: rng = pd.date_range("1/1/2012", periods=100, freq="S")
In [290]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
In [291]: ts.resample("5Min").sum()
Out[291]:
2012-01-01 25103
Freq: 5T, dtype: int64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency conversion and resampling operation.

Any function available via dispatching is available as a method of the returned object, including `sum`, `mean`, `std`, `sem`, `max`, `min`, `median`, `first`, `last`, `ohlc`:

```python
In [292]: ts.resample("5Min").mean()
Out[292]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

```python
In [293]: ts.resample("5Min").ohlc()
Out[293]:
     open  high  low  close
2012-01-01  308  460    9  205
```

```python
In [294]: ts.resample("5Min").max()
Out[294]:
2012-01-01  460
Freq: 5T, dtype: int64
```

For downsampling, `closed` can be set to ‘left’ or ‘right’ to specify which end of the interval is closed:
Parameters like `label` are used to manipulate the resulting labels. `label` specifies whether the result is labeled with the beginning or the end of the interval.

```
In [297]: ts.resample("5Min", label="left").mean()
Out[297]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

```
In [298]: ts.resample("5Min", label="left").mean()
Out[298]:
2012-01-01 251.03
Freq: 5T, dtype: float64
```

**Warning:** The default values for `label` and `closed` is ‘left’ for all frequency offsets except for ‘M’, ‘A’, ‘Q’, ‘BM’, ‘BA’, ‘BQ’, and ‘W’ which all have a default of ‘right’.

This might unintendedly lead to looking ahead, where the value for a later time is pulled back to a previous time as in the following example with the `BusinessDay` frequency:

```
In [299]: s = pd.date_range("2000-01-01", "2000-01-05").to_series()
In [300]: s.iloc[2] = pd.NaT
In [301]: s.dt.day_name()
Out[301]:
2000-01-01 Saturday
2000-01-02 Sunday
2000-01-03 NaN
2000-01-04 Tuesday
2000-01-05 Wednesday
Freq: D, dtype: object
```

```
# default: label='left', closed='left'
In [302]: s.resample("B").last().dt.day_name()
Out[302]:
1999-12-31 Sunday
2000-01-03 NaN
2000-01-04 Tuesday
2000-01-05 Wednesday
Freq: B, dtype: object
```

Notice how the value for Sunday got pulled back to the previous Friday. To get the behavior where the value for
Sunday is pushed to Monday, use instead

```python
In [303]: s.resample("B", label="right", closed="right").last().dt.day_name()
Out[303]:
2000-01-03 Sunday
2000-01-04 Tuesday
2000-01-05 Wednesday
Freq: B, dtype: object
```

The `axis` parameter can be set to 0 or 1 and allows you to resample the specified axis for a DataFrame.

`kind` can be set to ‘timestamp’ or ‘period’ to convert the resulting index to/from timestamp and time span representations. By default `resample` retains the input representation.

`convention` can be set to ‘start’ or ‘end’ when resampling period data (detail below). It specifies how low frequency periods are converted to higher frequency periods.

### Upsampling

For upsampling, you can specify a way to upsample and the `limit` parameter to interpolate over the gaps that are created:

```python
# from secondly to every 250 milliseconds
In [304]: ts[:2].resample("250L").asfreq()
Out[304]:
2012-01-01 00:00:00.000 308.0
2012-01-01 00:00:00.250 NaN
2012-01-01 00:00:00.500 NaN
2012-01-01 00:00:00.750 NaN
2012-01-01 00:00:01.000 204.0
Freq: 250L, dtype: float64

In [305]: ts[:2].resample("250L").ffill()
Out[305]:
2012-01-01 00:00:00.000 308
2012-01-01 00:00:00.250 308
2012-01-01 00:00:00.500 308
2012-01-01 00:00:00.750 308
2012-01-01 00:00:01.000 204
Freq: 250L, dtype: int64

In [306]: ts[:2].resample("250L").ffill(limit=2)
Out[306]:
2012-01-01 00:00:00.000 308.0
2012-01-01 00:00:00.250 308.0
2012-01-01 00:00:00.500 308.0
2012-01-01 00:00:00.750 NaN
2012-01-01 00:00:01.000 204.0
Freq: 250L, dtype: float64
```
Sparse resampling

Sparse timeseries are the ones where you have a lot fewer points relative to the amount of time you are looking to resample. Naively upsampling a sparse series can potentially generate lots of intermediate values. When you don't want to use a method to fill these values, e.g. fill_method is None, then intermediate values will be filled with NaN. Since resample is a time-based groupby, the following is a method to efficiently resample only the groups that are not all NaN.

```
In [307]: rng = pd.date_range("2014-1-1", periods=100, freq="D") + pd.Timedelta("1s")
In [308]: ts = pd.Series(range(100), index=rng)
```

If we want to resample to the full range of the series:

```
In [309]: ts.resample("3T").sum()
Out[309]:
2014-01-01 00:00:00   0
2014-01-01 00:03:00   0
2014-01-01 00:06:00   0
2014-01-01 00:09:00   0
2014-01-01 00:12:00   0
   ...                  ...
2014-04-09 23:48:00   0
2014-04-09 23:51:00   0
2014-04-09 23:54:00   0
2014-04-09 23:57:00   0
2014-04-10 00:00:00   99
Freq: 3T, Length: 47521, dtype: int64
```

We can instead only resample those groups where we have points as follows:

```
In [310]: from functools import partial
In [311]: from pandas.tseries.frequencies import to_offset
In [312]: def round(t, freq):
   ....:     freq = to_offset(freq)
   ....:     return pd.Timestamp((t.value // freq.delta.value) * freq.delta.value)
   ....:
In [313]: ts.groupby(partial(round, freq="3T")).sum()
Out[313]:
2014-01-01   0
2014-01-02   1
2014-01-03   2
2014-01-04   3
2014-01-05   4
   ...      ...
2014-04-06  95
2014-04-07  96
2014-04-08  97
2014-04-09  98
2014-04-10  99
Length: 100, dtype: int64
```
Aggregation

Similar to the aggregating API, groupby API, and the window API, a Resampler can be selectively resampled.

Resampling a DataFrame, the default will be to act on all columns with the same function.

```
In [314]: df = pd.DataFrame(
      ....:     np.random.randn(1000, 3),
      ....:     index=pd.date_range("1/1/2012", freq="S", periods=1000),
      ....:     columns=["A", "B", "C"],
      ....:     )
      ....:
In [315]: r = df.resample("3T")
In [316]: r.mean()
Out[316]:
     A        B        C
2012-01-01 00:00:00 -0.033823 -0.121514 -0.081447
2012-01-01 00:03:00  0.056909  0.146731 -0.024320
2012-01-01 00:06:00 -0.058837  0.047046 -0.052021
2012-01-01 00:09:00  0.063123 -0.026158 -0.066533
2012-01-01 00:12:00  0.186340 -0.003144  0.074752
2012-01-01 00:15:00 -0.085954 -0.016287 -0.050046
```

We can select a specific column or columns using standard getitem.

```
In [317]: r["A"].mean()
Out[317]:
2012-01-01 00:00:00 -0.033823
2012-01-01 00:03:00  0.056909
2012-01-01 00:06:00 -0.058837
2012-01-01 00:09:00  0.063123
2012-01-01 00:12:00  0.186340
2012-01-01 00:15:00 -0.085954
Freq: 3T, Name: A, dtype: float64
```

```
In [318]: r["A", "B"].mean()
Out[318]:
     A        B
2012-01-01 00:00:00 -0.033823 -0.121514
2012-01-01 00:03:00  0.056909  0.146731
2012-01-01 00:06:00 -0.058837  0.047046
2012-01-01 00:09:00  0.063123 -0.026158
2012-01-01 00:12:00  0.186340 -0.003144
2012-01-01 00:15:00 -0.085954 -0.016287
```

You can pass a list or dict of functions to do aggregation with, outputting a DataFrame:

```
In [319]: r["A"].agg([np.sum, np.mean, np.std])
Out[319]:
         sum  mean   std
2012-01-01 00:00:00 -6.088060 -0.033823  1.043263
2012-01-01 00:03:00 10.243678  0.056909  1.058534
```

(continues on next page)
On a resampled DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
2012-01-01 00:00:00 -6.088060 -0.033823 -21.872530 -0.121514 -14.660515 -0.081447
2012-01-01 00:03:00 10.243678 0.056909 26.411633 0.146731 -4.377642 -0.024320
2012-01-01 00:06:00 -10.590584 -0.085954 8.468289 0.047046 -9.363825 -0.052021
2012-01-01 00:09:00 11.362228 0.063123 -4.708526 -0.026158 -11.975895 -0.066533
2012-01-01 00:12:00 33.541257 0.186340 -0.565895 -0.003144 13.455299 0.074752
2012-01-01 00:15:00 -8.595393 -0.085954 -1.628689 -0.016287 -5.004580 -0.050046
```

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```
A               B
2012-01-01 00:00:00 -6.088060 1.001294
2012-01-01 00:03:00 10.243678 1.074597
2012-01-01 00:06:00 -10.590584 0.987309
2012-01-01 00:09:00 11.362228 0.944953
2012-01-01 00:12:00 33.541257 1.095025
2012-01-01 00:15:00 -8.595393 1.035312
```

The function names can also be strings. In order for a string to be valid it must be implemented on the resampled object:

```
A               B
2012-01-01 00:00:00 -6.088060 1.001294
2012-01-01 00:03:00 10.243678 1.074597
2012-01-01 00:06:00 -10.590584 0.987309
2012-01-01 00:09:00 11.362228 0.944953
2012-01-01 00:12:00 33.541257 1.095025
2012-01-01 00:15:00 -8.595393 1.035312
```

Furthermore, you can also specify multiple aggregation functions for each column separately.

```
A sum std B mean std
2012-01-01 00:00:00 -6.088060 1.043263 -0.121514 1.001294
2012-01-01 00:03:00 10.243678 1.058534 0.146731 1.074597
2012-01-01 00:06:00 -10.590584 0.949264 0.047046 0.987309
2012-01-01 00:09:00 11.362228 1.028096 -0.026158 0.944953
```
If a DataFrame does not have a datetimelike index, but instead you want to resample based on datetimelike column in the frame, it can passed to the on keyword.

```
In [324]: df = pd.DataFrame(
    ...:     {"date": pd.date_range("2015-01-01", freq="W", periods=5), "a": np.arange(5)},
    ...:     index=pd.MultiIndex.from_arrays(
    ...:     [[1, 2, 3, 4, 5], pd.date_range("2015-01-01", freq="W", periods=5)],
    ...:     names=["v", "d"],
    ...:     )
    ...
```

```
In [325]: df
Out[325]:
   date          a
   v      d
1 2015-01-04 2015-01-04 0
2 2015-01-11 2015-01-11 1
3 2015-01-18 2015-01-18 2
4 2015-01-25 2015-01-25 3
5 2015-02-01 2015-02-01 4
```

```
In [326]: df.resample("M", on="date").sum()
Out[326]:
   a
   date
2015-01-31 6
2015-02-28 4
```

Similarly, if you instead want to resample by a datetimelike level of MultiIndex, its name or location can be passed to the level keyword.

```
In [327]: df.resample("M", level="d").sum()
Out[327]:
   a
d
2015-01-31 6
2015-02-28 4
```
Iterating through groups

With the Resampler object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```python
In [328]: small = pd.Series(
       ...:     range(6),
       ...:     index=pd.to_datetime(
       ...:     [
       ...:         "2017-01-01T00:00:00",
       ...:         "2017-01-01T00:30:00",
       ...:         "2017-01-01T00:31:00",
       ...:         "2017-01-01T01:00:00",
       ...:         "2017-01-01T03:00:00",
       ...:         "2017-01-01T03:05:00",
       ...:     ]
       ...: ),
       ...: )

In [329]: resampled = small.resample("H")

In [330]: for name, group in resampled:
       ...:     print("Group: ", name)
       ...:     print("-----------")
       ...:     print(group, end="\n")

Group: 2017-01-01 00:00:00
-----------
2017-01-01 00:00:00 0
2017-01-01 00:30:00 1
2017-01-01 00:31:00 2
dtype: int64

Group: 2017-01-01 01:00:00
-----------
2017-01-01 01:00:00 3
dtype: int64

Group: 2017-01-01 02:00:00
-----------
Series([], dtype: int64)

Group: 2017-01-01 03:00:00
-----------
2017-01-01 03:00:00 4
2017-01-01 03:05:00 5
dtype: int64
```

See *Iterating through groups* or `Resampler.__iter__` for more.
Use `origin` or `offset` to adjust the start of the bins

New in version 1.1.0.

The bins of the grouping are adjusted based on the beginning of the day of the time series starting point. This works well with frequencies that are multiples of a day (like `30D`) or that divide a day evenly (like `90s` or `1min`). This can create inconsistencies with some frequencies that do not meet this criteria. To change this behavior you can specify a fixed Timestamp with the argument `origin`.

For example:

```python
In [331]: start, end = "2000-10-01 23:30:00", "2000-10-02 00:30:00"
In [332]: middle = "2000-10-02 00:00:00"
In [333]: rng = pd.date_range(start, end, freq="7min")
In [334]: ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
```

```
In [335]: ts
Out[335]:
2000-10-01 23:30:00    0
2000-10-01 23:37:00    3
2000-10-01 23:44:00    6
2000-10-01 23:51:00    9
2000-10-01 23:58:00   12
2000-10-02 00:05:00   15
2000-10-02 00:12:00   18
2000-10-02 00:19:00   21
2000-10-02 00:26:00   24
Freq: 7T, dtype: int64
```

Here we can see that, when using `origin` with its default value (`'start_day'`), the result after `2000-10-02 00:00:00` are not identical depending on the start of time series:

```python
In [336]: ts.resample("17min", origin="start_day").sum()
Out[336]:
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
2000-10-02 00:22:00   24
Freq: 17T, dtype: int64
```

Here we can see that, when setting `origin` to `epoch`, the result after `2000-10-02 00:00:00` are identical depending on the start of time series:

```python
In [337]: ts[middle:end].resample("17min", origin="start_day").sum()
Out[337]:
2000-10-02 00:00:00   33
2000-10-02 00:17:00   45
Freq: 17T, dtype: int64
```

(continues on next page)
In [339]: ts[middle:end].resample("17min", origin="epoch").sum()
Out[339]:
2000-10-01 23:18:00    0
2000-10-01 23:35:00    18
2000-10-01 23:52:00    27
2000-10-02 00:09:00    39
2000-10-02 00:26:00    24
Freq: 17T, dtype: int64

If needed you can use a custom timestamp for origin:

In [340]: ts.resample("17min", origin="2001-01-01").sum()
Out[340]:
2000-10-01 23:30:00    9
2000-10-01 23:47:00   21
2000-10-02 00:04:00   54
2000-10-02 00:21:00   24
Freq: 17T, dtype: int64

In [341]: ts[middle:end].resample("17min", origin=pd.Timestamp("2001-01-01")).sum()
Out[341]:
2000-10-02 00:04:00   54
2000-10-02 00:21:00   24
Freq: 17T, dtype: int64

If needed you can just adjust the bins with an offset Timedelta that would be added to the default origin. Those two examples are equivalent for this time series:

In [342]: ts.resample("17min", origin="start").sum()
Out[342]:
2000-10-01 23:30:00    9
2000-10-01 23:47:00   21
2000-10-02 00:04:00   54
2000-10-02 00:21:00   24
Freq: 17T, dtype: int64

In [343]: ts.resample("17min", offset="23h30min").sum()
Out[343]:
2000-10-01 23:30:00    9
2000-10-01 23:47:00   21
2000-10-02 00:04:00   54
2000-10-02 00:21:00   24
Freq: 17T, dtype: int64

Note the use of 'start' for origin on the last example. In that case, origin will be set to the first value of the timeseries.

2.20. Time series / date functionality 889
Backward resample

New in version 1.3.0.

Instead of adjusting the beginning of bins, sometimes we need to fix the end of the bins to make a backward resample with a given freq. The backward resample sets closed to 'right' by default since the last value should be considered as the edge point for the last bin.

We can set origin to 'end'. The value for a specific Timestamp index stands for the resample result from the current Timestamp minus freq to the current Timestamp with a right close.

```python
In [344]: ts.resample('17min', origin='end').sum()
Out[344]:
2000-10-01 23:35:00 0
2000-10-01 23:52:00 18
2000-10-02 00:09:00 27
2000-10-02 00:26:00 63
Freq: 17T, dtype: int64
```

Besides, in contrast with the 'start_day' option, end_day is supported. This will set the origin as the ceiling midnight of the largest Timestamp.

```python
In [345]: ts.resample('17min', origin='end_day').sum()
Out[345]:
2000-10-01 23:38:00 3
2000-10-01 23:55:00 15
2000-10-02 00:12:00 45
2000-10-02 00:29:00 45
Freq: 17T, dtype: int64
```

The above result uses 2000-10-02 00:29:00 as the last bin's right edge since the following computation.

```python
In [346]: ceil_mid = rng.max().ceil('D')
In [347]: freq = pd.offsets.Minute(17)
In [348]: bin_res = ceil_mid - freq * ((ceil_mid - rng.max()) // freq)
In [349]: bin_res
Out[349]: Timestamp('2000-10-02 00:29:00')
```
2.20.11 Time span representation

Regular intervals of time are represented by `Period` objects in pandas while sequences of `Period` objects are collected in a `PeriodIndex`, which can be created with the convenience function `period_range`.

**Period**

A `Period` represents a span of time (e.g., a day, a month, a quarter, etc). You can specify the span via `freq` keyword using a frequency alias like below. Because `freq` represents a span of `Period`, it cannot be negative like “-3D”.

```
In [350]: pd.Period("2012", freq="A-DEC")
Out[350]: Period('2012', 'A-DEC')

In [351]: pd.Period("2012-1-1", freq="D")
Out[351]: Period('2012-01-01', 'D')

In [352]: pd.Period("2012-1-1 19:00", freq="H")
Out[352]: Period('2012-01-01 19:00', 'H')

In [353]: pd.Period("2012-1-1 19:00", freq="5H")
Out[353]: Period('2012-01-01 19:00', '5H')
```

Adding and subtracting integers from periods shifts the period by its own frequency. Arithmetic is not allowed between `Period` with different `freq` (span).

```
In [354]: p = pd.Period("2012", freq="A-DEC")

In [355]: p + 1
Out[355]: Period('2013', 'A-DEC')

In [356]: p - 3
Out[356]: Period('2009', 'A-DEC')

In [357]: p = pd.Period("2012-01", freq="2M")

In [358]: p + 2
Out[358]: Period('2012-05', '2M')

In [359]: p - 1
Out[359]: Period('2011-11', '2M')

In [360]: p == pd.Period("2012-01", freq="3M")
Out[360]: False
```

If `Period` `freq` is daily or higher (D, H, T, S, L, U, N), offsets and `timedelta`-like can be added if the result can have the same freq. Otherwise, `ValueError` will be raised.

```
In [361]: p = pd.Period("2014-07-01 09:00", freq="H")

In [362]: p + pd.offsets.Hour(2)
Out[362]: Period('2014-07-01 11:00', 'H')

In [363]: p + datetime.timedelta(minutes=120)
```
Out[363]: Period('2014-07-01 11:00', 'H')

In [364]: p + np.timedelta64(7200, "s")
Out[364]: Period('2014-07-01 11:00', 'H')

In [1]: p + pd.offsets.Minute(5)
Traceback
...
ValueError: Input has different freq from Period(freq=H)

If Period has other frequencies, only the same offsets can be added. Otherwise, ValueError will be raised.

In [365]: p = pd.Period("2014-07", freq="M")

In [366]: p + pd.offsets.MonthEnd(3)
Out[366]: Period('2014-10', 'M')

In [1]: p + pd.offsets.MonthBegin(3)
Traceback
...
ValueError: Input has different freq from Period(freq=M)

Taking the difference of Period instances with the same frequency will return the number of frequency units between them:

Out[367]: <10 * YearEnds: month=12>

PeriodIndex and period_range

Regular sequences of Period objects can be collected in a PeriodIndex, which can be constructed using the period_range convenience function:

In [368]: prng = pd.period_range("1/1/2011", "1/1/2012", freq="M")

In [369]: prng

The PeriodIndex constructor can also be used directly:

In [370]: pd.PeriodIndex(['2011-1', '2011-2', '2011-3'], freq="M")
Out[370]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]')

Passing multiplied frequency outputs a sequence of Period which has multiplied span.

In [371]: pd.period_range(start="2014-01", freq="3M", periods=4)
If `start` or `end` are `Period` objects, they will be used as anchor endpoints for a `PeriodIndex` with frequency matching that of the `PeriodIndex` constructor.

```
In [372]: pd.period_range(
    .....:     start=pd.Period("2017Q1", freq="Q"), end=pd.Period("2017Q2", freq="Q"),
        freq="M"
    .....:)
Out[372]: PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'], dtype='period[M]')
```

Just like `DatetimeIndex`, a `PeriodIndex` can also be used to index pandas objects:

```
In [373]: ps = pd.Series(np.random.randn(len(prng)), prng)
In [374]: ps
Out[374]:
2011-01  -2.916901
2011-02   0.514474
2011-03   1.346470
2011-04   0.816397
2011-05   2.258648
2011-06   0.494789
2011-07   0.301239
2011-08   0.464776
2011-09  -1.393581
2011-10   0.056780
2011-11   0.197035
2011-12   2.261385
2012-01  -0.329583
Freq: M, dtype: float64
```

`PeriodIndex` supports addition and subtraction with the same rule as `Period`.

```
In [375]: idx = pd.period_range("2014-07-01 09:00", periods=5, freq="H")
In [376]: idx
Out[376]:
PeriodIndex(['2014-07-01 09:00', '2014-07-01 10:00', '2014-07-01 11:00',
             '2014-07-01 12:00', '2014-07-01 13:00'],
            dtype='period[H]')
In [377]: idx + pd.offsets.Hour(2)
Out[377]:
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
             '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]')
In [378]: idx = pd.period_range("2014-07", periods=5, freq="M")
In [379]: idx
Out[379]:
            dtype='period[M]')
In [380]: idx + pd.offsets.MonthEnd(3)
```
(continues on next page)
PeriodIndex has its own dtype named period, refer to *Period Dtypes*.

**Period dtypes**

PeriodIndex has a custom period dtype. This is a pandas extension dtype similar to the *timezone aware dtype* (datetime64[ns, tz]).

The period dtype holds the `freq` attribute and is represented with period[freq] like period[D] or period[M], using *frequency strings*.

```python
In [381]: pi = pd.period_range("2016-01-01", periods=3, freq="M")

In [382]: pi
Out[382]: PeriodIndex(['2016-01', '2016-02', '2016-03'], dtype='period[M]')

In [383]: pi.dtype
Out[383]: period[M]
```

The period dtype can be used in `.astype(...)`. It allows one to change the `freq` of a PeriodIndex like `.asfreq()` and convert a DatetimeIndex to PeriodIndex like `.to_period()`:

```python
# change monthly freq to daily freq
In [384]: pi.astype("period[D"])
Out[384]: PeriodIndex(['2016-01-31', '2016-02-29', '2016-03-31'], dtype='period[D]')

# convert to DatetimeIndex
In [385]: pi.astype("datetime64[ns"]
Out[385]: DatetimeIndex(['2016-01-01', '2016-02-01', '2016-03-01'], dtype='datetime64[ns]'
              , freq='MS')

# convert to PeriodIndex
In [386]: dti = pd.date_range("2011-01-01", freq="M", periods=3)

In [387]: dti
Out[387]: DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31'], dtype='datetime64[ns]'
              , freq='M')

In [388]: dti.astype("period[M"]
Out[388]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]')
```
**PeriodIndex partial string indexing**

PeriodIndex now supports partial string slicing with non-monotonic indexes.

New in version 1.1.0.

You can pass in dates and strings to `Series` and `DataFrame` with `PeriodIndex`, in the same manner as `DatetimeIndex`. For details, refer to *DatetimeIndex Partial String Indexing*.

```
In [389]: ps["2011-01"]
Out[389]: -2.9169013294054507

In [390]: ps[datetime.datetime(2011, 12, 25):]
Out[390]:
2011-12  2.261385
2012-01  -0.329583
Freq: M, dtype: float64

In [391]: ps["10/31/2011":"12/31/2011"]
Out[391]:
2011-10  0.056780
2011-11  0.197035
2011-12  2.261385
Freq: M, dtype: float64
```

Passing a string representing a lower frequency than `PeriodIndex` returns partial sliced data.

```
In [392]: ps["2011"]
Out[392]:
2011-01  -2.916901
2011-02  0.514474
2011-03  1.346470
2011-04  0.816397
2011-05  2.258648
2011-06  0.494789
2011-07  0.301239
2011-08  0.464776
2011-09  1.393581
2011-10  0.056780
2011-11  0.197035
2011-12  2.261385
Freq: M, dtype: float64

In [393]: dfp = pd.DataFrame(
       ....:     np.random.randn(600, 1),
       ....:     columns=["A"],
       ....:     index=pd.period_range("2013-01-01 9:00", periods=600, freq="T"),
       ....:     )

In [394]: dfp
Out[394]:
  A
2013-01-01 09:00  -0.538468
2013-01-01 09:01  -1.365819
```

(continues on next page)
As with `DatetimeIndex`, the endpoints will be included in the result. The example below slices data starting from 10:00 to 11:59.

```python
In [396]: dfp['2013-01-01 10H':'2013-01-01 11H']
Out[396]:
    A
2013-01-01 10:00  -0.308975
2013-01-01 10:01   0.542520
2013-01-01 10:02   1.061068
2013-01-01 10:03   0.754005
2013-01-01 10:04   0.352933
... ...                      ...
2013-01-01 11:55  -0.590204
2013-01-01 11:56   1.539990
2013-01-01 11:57  -1.224826
2013-01-01 11:58   0.578798
2013-01-01 11:59  -0.685496
[120 rows x 1 columns]
```
**Frequency conversion and resampling with PeriodIndex**

The frequency of `Period` and `PeriodIndex` can be converted via the `asfreq` method. Let’s start with the fiscal year 2011, ending in December:

```python
In [397]: p = pd.Period("2011", freq="A-DEC")
```

```python
In [398]: p
Out[398]: Period('2011', 'A-DEC')
```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```python
In [399]: p.asfreq("M", how="start")
Out[399]: Period('2011-01', 'M')
```

```python
In [400]: p.asfreq("M", how="end")
Out[400]: Period('2011-12', 'M')
```

The shorthands ‘s’ and ‘e’ are provided for convenience:

```python
In [401]: p.asfreq("M", "s")
Out[401]: Period('2011-01', 'M')
```

```python
In [402]: p.asfreq("M", "e")
Out[402]: Period('2011-12', 'M')
```

Converting to a “super-period” (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```python
In [403]: p = pd.Period("2011-12", freq="M")
```

```python
In [404]: p.asfreq("A-NOV")
Out[404]: Period('2012', 'A-NOV')
```

Note that since we converted to an annual frequency that ends the year in November, the monthly period of December 2011 is actually in the 2012 A-NOV period.

Period conversions with anchored frequencies are particularly useful for working with various quarterly data common to economics, business, and other fields. Many organizations define quarters relative to the month in which their fiscal year starts and ends. Thus, first quarter of 2011 could start in 2010 or a few months into 2011. Via anchored frequencies, pandas works for all quarterly frequencies Q-JAN through Q-DEC.

Q-DEC defines regular calendar quarters:

```python
In [405]: p = pd.Period("2012Q1", freq="Q-DEC")
```

```python
In [406]: p.asfreq("Q-DEC")
Out[406]: Period('2012-01', 'Q-DEC')
```

```python
In [407]: p.asfreq("Q-DEC")
Out[407]: Period('2012-03', 'Q-DEC')
```

Q-MAR defines fiscal year end in March:
2.20.12 Converting between representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```python
In [411]: rng = pd.date_range("1/1/2012", periods=5, freq="M")
In [412]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [413]: ts
Out[413]:
2012-01-31  1.931253
2012-02-29  -0.184594
2012-03-31   0.249656
2012-04-30  -0.978151
2012-05-31  -0.873389
Freq: M, dtype: float64
In [414]: ps = ts.to_period()
In [415]: ps
Out[415]:
2012-01  1.931253
2012-02  -0.184594
2012-03   0.249656
2012-04  -0.978151
2012-05  -0.873389
Freq: M, dtype: float64
In [416]: ps.to_timestamp()
Out[416]:
2012-01-01  1.931253
2012-02-01  -0.184594
2012-03-01   0.249656
2012-04-01  -0.978151
2012-05-01  -0.873389
Freq: MS, dtype: float64
```

Remember that ‘s’ and ‘e’ can be used to return the timestamps at the start or end of the period:

```python
In [417]: ps.to_timestamp("D", how="s")
Out[417]:
2012-01-01  1.931253
2012-02-01  -0.184594
2012-03-01   0.249656
```
Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```python
In [418]: prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")
In [419]: ts = pd.Series(np.random.randn(len(prng)), prng)
In [420]: ts.index = (prng.asfreq("M", "e") + 1).asfreq("H", "s") + 9
In [421]: ts.head()
Out[421]:
1990-03-01 09:00 -0.109291
1990-06-01 09:00 -0.637235
1990-09-01 09:00 -1.735925
1990-12-01 09:00 2.096946
1991-03-01 09:00 -1.039926
Freq: H, dtype: float64
```

### 2.2.13 Representing out-of-bounds spans

If you have data that is outside of the `Timestamp` bounds, see *Timestamp limitations*, then you can use a `PeriodIndex` and/or `Series` of `Periods` to do computations.

```python
In [422]: span = pd.period_range("1215-01-01", "1381-01-01", freq="D")
In [423]: span
Out[423]:
PeriodIndex(['1215-01-01', '1215-01-02', '1215-01-03', '1215-01-04',
 '1215-01-05', '1215-01-06', '1215-01-07', '1215-01-08',
 '1215-01-09', '1215-01-10',
 ..., '1380-12-23', '1380-12-24', '1380-12-25', '1380-12-26',
 '1380-12-27', '1380-12-28', '1380-12-29', '1380-12-30',
 '1380-12-31', '1381-01-01'], dtype='period[D]', length=60632)
```

To convert from an `int64` based YYYYMMDD representation.

```python
In [424]: s = pd.Series([20121231, 20141130, 99991231])
In [425]: s
Out[425]:
0    20121231
1    20141130
2   99991231
dtype: int64
```
In [426]: ```python
def conv(x):
    ......:    return pd.Period(year=x // 10000, month=x // 100 % 100, day=x % 100, freq="D")
    ......:
```
In [427]: s.apply(conv)
Out[427]:
0  2012-12-31
1  2014-11-30
2  9999-12-31
dtype: period[D]
In [428]: s.apply(conv)[2]
Out[428]: Period('9999-12-31', 'D')

These can easily be converted to a `PeriodIndex`:

In [429]: span = pd.PeriodIndex(s.apply(conv))

In [430]: span
Out[430]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='period[D]')

## 2.20.14 Time zone handling

pandas provides rich support for working with timestamps in different time zones using the `pytz` and `dateutil` libraries or `datetime.timezone` objects from the standard library.

### Working with time zones

By default, pandas objects are time zone unaware:

In [431]: rng = pd.date_range("3/6/2012 00:00", periods=15, freq="D")

In [432]: rng.tz is None
Out[432]: True

To localize these dates to a time zone (assign a particular time zone to a naive date), you can use the `tz_localize` method or the `tz` keyword argument in `date_range()`, `Timestamp`, or `DatetimeIndex`. You can either pass `pytz` or `dateutil` time zone objects or Olson time zone database strings. Olson time zone strings will return `pytz` time zone objects by default. To return `dateutil` time zone objects, append `dateutil/` before the string.

- In `pytz` you can find a list of common (and less common) time zones using `from pytz import common_timezones, all_timezones`.
- `dateutil` uses the OS time zones so there isn’t a fixed list available. For common zones, the names are the same as `pytz`.

In [433]: ```python
import dateutil
# pytz
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [434]: rng_pytz = pd.date_range("3/6/2012 00:00", periods=3, freq="D", tz="Europe/London")

In [435]: rng_pytz.tz
Out[435]: <DstTzInfo 'Europe/London' LMT-1 day, 23:59:00 STD>

# dateutil

In [436]: rng_dateutil = pd.date_range("3/6/2012 00:00", periods=3, freq="D")

In [437]: rng_dateutil = rng_dateutil.tz_localize("dateutil/Europe/London")

In [438]: rng_dateutil.tz
Out[438]: tzfile('/usr/share/zoneinfo/Europe/London')

# dateutil - utc special case

In [439]: rng_utc = pd.date_range("3/6/2012 00:00", periods=3, freq="D", tz=dateutil.tz.tzutc(), tz=datatime.timezone.utc,

In [440]: rng_utc.tz
Out[440]: tzutc()

New in version 0.25.0.

# datetime.timezone

In [441]: rng_utc = pd.date_range("3/6/2012 00:00", periods=3, freq="D", tz=datetime.timezone.utc,

In [442]: rng_utc.tz
Out[442]: datatime.timezone.utc

Note that the UTC time zone is a special case in dateutil and should be constructed explicitly as an instance of dateutil.tz.tzutc. You can also construct other time zones objects explicitly first.

In [443]: import pytz

# pytz

In [444]: tz_pytz = pytz.timezone("Europe/London")

In [445]: rng_pytz = pd.date_range("3/6/2012 00:00", periods=3, freq="D")

In [446]: rng_pytz = rng_pytz.tz_localize(tz_pytz)
To convert a time zone aware pandas object from one time zone to another, you can use the `tz_convert` method.

```python
In [451]: rng_pytz.tz_convert("US/Eastern")
Out[451]:
DatetimeIndex(['2012-03-05 19:00:00-05:00', '2012-03-06 19:00:00-05:00',
              '2012-03-07 19:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

**Note:** When using pytz time zones, `DatetimeIndex` will construct a different time zone object than a `Timestamp` for the same time zone input. A `DatetimeIndex` can hold a collection of `Timestamp` objects that may have different UTC offsets and cannot be succinctly represented by one pytz time zone instance while one `Timestamp` represents one point in time with a specific UTC offset.

```python
In [452]: dti = pd.date_range("2019-01-01", periods=3, freq="D", tz="US/Pacific")
```

```python
In [453]: dti.tz
Out[453]: <DstTzInfo 'US/Pacific' LMT-1 day, 16:07:00 STD>
```

```python
In [454]: ts = pd.Timestamp("2019-01-01", tz="US/Pacific")
```

```python
In [455]: ts.tz
Out[455]: <DstTzInfo 'US/Pacific' PST-1 day, 16:00:00 STD>
```

**Warning:** Be wary of conversions between libraries. For some time zones, pytz and dateutil have different definitions of the zone. This is more of a problem for unusual time zones than for 'standard' zones like US/Eastern.

**Warning:** Be aware that a time zone definition across versions of time zone libraries may not be considered equal. This may cause problems when working with stored data that is localized using one version and operated on with a different version. See [here](#) for how to handle such a situation.

**Warning:** For pytz time zones, it is incorrect to pass a time zone object directly into the `datetime.datetime` constructor (e.g., `datetime.datetime(2011, 1, 1, tzinfo=pytz.timezone('US/Eastern'))`). Instead, the datetime needs to be localized using the `localize` method on the pytz time zone object.
**Warning:** Be aware that for times in the future, correct conversion between time zones (and UTC) cannot be guaranteed by any time zone library because a timezone’s offset from UTC may be changed by the respective government.

**Warning:** If you are using dates beyond 2038-01-18, due to current deficiencies in the underlying libraries caused by the year 2038 problem, daylight saving time (DST) adjustments to timezone aware dates will not be applied. If and when the underlying libraries are fixed, the DST transitions will be applied.

For example, for two dates that are in British Summer Time (and so would normally be GMT+1), both the following asserts evaluate as true:

```python
In [456]: d_2037 = "2037-03-31T010101"
In [457]: d_2038 = "2038-03-31T010101"
In [458]: DST = "Europe/London"
In [459]: assert pd.Timestamp(d_2037, tz=DST) != pd.Timestamp(d_2037, tz="GMT")
In [460]: assert pd.Timestamp(d_2038, tz=DST) == pd.Timestamp(d_2038, tz="GMT")
```

Under the hood, all timestamps are stored in UTC. Values from a time zone aware `DatetimeIndex` or `Timestamp` will have their fields (day, hour, minute, etc.) localized to the time zone. However, timestamps with the same UTC value are still considered to be equal even if they are in different time zones:

```python
In [461]: rng_eastern = rng_utc.tz_convert("US/Eastern")
In [462]: rng_berlin = rng_utc.tz_convert("Europe/Berlin")
In [463]: rng_eastern[2]
Out[463]: Timestamp('2012-03-07 19:00:00-0500', tz='US/Eastern', freq='D')
In [464]: rng_berlin[2]
Out[464]: Timestamp('2012-03-08 01:00:00+0100', tz='Europe/Berlin', freq='D')
Out[465]: True
```

Operations between `Series` in different time zones will yield UTC `Series`, aligning the data on the UTC timestamps:

```python
In [466]: ts_utc = pd.Series(range(3), pd.date_range("20130101", periods=3, tz="UTC"))
In [467]: eastern = ts_utc.tz_convert("US/Eastern")
In [468]: berlin = ts_utc.tz_convert("Europe/Berlin")
In [469]: result = eastern + berlin
In [470]: result
```

(continues on next page)
To remove time zone information, use `tz_localize(None)` or `tz_convert(None)`.

`tz_localize(None)` will remove the time zone yielding the local time representation. `tz_convert(None)` will remove the time zone after converting to UTC time.

```python
In [472]: didx = pd.date_range(start="2014-08-01 09:00", freq="H", periods=3, tz="US/Eastern")

In [473]: didx
Out[473]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
              '2014-08-01 11:00:00-04:00'],
dtype='datetime64[ns, US/Eastern]', freq='H')

In [474]: didx.tz_localize(None)
Out[474]:
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
              '2014-08-01 11:00:00'],
dtype='datetime64[ns]', freq=None)

In [475]: didx.tz_convert(None)
Out[475]:
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
              '2014-08-01 15:00:00'],
dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical to tz_convert('UTC').tz_localize(None)

In [476]: didx.tz_convert("UTC").tz_localize(None)
Out[476]:
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
              '2014-08-01 15:00:00'],
dtype='datetime64[ns]', freq=None)
Fold

New in version 1.1.0.

For ambiguous times, pandas supports explicitly specifying the keyword-only fold argument. Due to daylight saving time, one wall clock time can occur twice when shifting from summer to winter time; fold describes whether the datetime-like corresponds to the first (0) or the second time (1) the wall clock hits the ambiguous time. Fold is supported only for constructing from naive `datetime.datetime` (see `datetime` documentation for details) or from `Timestamp` or for constructing from components (see below). Only `dateutil` timezones are supported (see `dateutil` documentation for dateutil methods that deal with ambiguous datetimes) as `pytz` timezones do not support fold (see `pytz` documentation for details on how `pytz` deals with ambiguous datetimes). To localize an ambiguous datetime with `pytz`, please use `Timestamp.tz_localize()`. In general, we recommend to rely on `Timestamp.tz_localize()` when localizing ambiguous datetimes if you need direct control over how they are handled.

```python
In [477]: pd.Timestamp(
    ...:     datetime.datetime(2019, 10, 27, 1, 30, 0, 0),
    ...:     tz="dateutil/Europe/London",
    ...:     fold=0,
    ...: )
```

```python
Out[477]: Timestamp('2019-10-27 01:30:00+0100', tz='dateutil//usr/share/zoneinfo/Europe/London')
```

```python
In [478]: pd.Timestamp(
    ...:     year=2019,
    ...:     month=10,
    ...:     day=27,
    ...:     hour=1,
    ...:     minute=30,
    ...:     tz="dateutil/Europe/London",
    ...:     fold=1,
    ...: )
```

```python
Out[478]: Timestamp('2019-10-27 01:30:00+0000', tz='dateutil//usr/share/zoneinfo/Europe/London')
```

Ambiguous times when localizing

tz_localize may not be able to determine the UTC offset of a timestamp because daylight savings time (DST) in a local time zone causes some times to occur twice within one day (“clocks fall back”). The following options are available:

- 'raise': Raises a `pytz.AmbiguousTimeError` (the default behavior)
- 'infer': Attempt to determine the correct offset base on the monotonicity of the timestamps
- 'NaT': Replaces ambiguous times with NaT
- bool: True represents a DST time, False represents non-DST time. An array-like of bool values is supported for a sequence of times.

```python
In [479]: rng_hourly = pd.DatetimeIndex(
    ...:     ["11/06/2011 00:00", "11/06/2011 01:00", "11/06/2011 01:00", "11/06/2011␣
    ...:     02:00"])
```

(continues on next page)
This will fail as there are ambiguous times ('11/06/2011 01:00')

```python
In [2]: rng_hourly.tz_localize('US/Eastern')
AmbiguousTimeError: Cannot infer dst time from Timestamp('2011-11-06 01:00:00'), try using the 'ambiguous' argument
```

Handle these ambiguous times by specifying the following.

```python
In [480]: rng_hourly.tz_localize("US/Eastern", ambiguous="infer")
Out[480]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', '2011-11-06 01:00:00-04:00',
              '2011-11-06 01:00:00-05:00', '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

```python
In [481]: rng_hourly.tz_localize("US/Eastern", ambiguous="NaT")
Out[481]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', 'NaT', 'NaT',
              '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

```python
In [482]: rng_hourly.tz_localize("US/Eastern", ambiguous=[True, True, False, False])
Out[482]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', '2011-11-06 01:00:00-04:00',
              '2011-11-06 01:00:00-05:00', '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

Nonexistent times when localizing

A DST transition may also shift the local time ahead by 1 hour creating nonexistent local times (“clocks spring forward”). The behavior of localizing a timeseries with nonexistent times can be controlled by the `nonexistent` argument. The following options are available:

- 'raise': Raises a `pytz.NonExistentTimeError` (the default behavior)
- 'NaT': Replaces nonexistent times with `NaT`
- 'shift_forward': Shifts nonexistent times forward to the closest real time
- 'shift_backward': Shifts nonexistent times backward to the closest real time
- timedelta object: Shifts nonexistent times by the timedelta duration

```python
In [483]: dti = pd.date_range(start="2015-03-29 02:30:00", periods=3, freq="H")
```

# 2:30 is a nonexistent time

Localization of nonexistent times will raise an error by default.

```python
In [2]: dti.tz_localize('Europe/Warsaw')
NonExistentTimeError: 2015-03-29 02:30:00
```

Transform nonexistent times to `NaT` or shift the times.
In [484]: dti
Out[484]:
DatetimeIndex(['2015-03-29 02:30:00', '2015-03-29 03:30:00',
              '2015-03-29 04:30:00'],
       dtype='datetime64[ns]', freq='H')

In [485]: dti.tz_localize("Europe/Warsaw", nonexistent="shift_forward")
Out[485]:
DatetimeIndex(['2015-03-29 03:00:00+02:00', '2015-03-29 03:30:00+02:00',
              '2015-03-29 04:30:00+02:00'],
       dtype='datetime64[ns, Europe/Warsaw]', freq=None)

In [486]: dti.tz_localize("Europe/Warsaw", nonexistent="shift_backward")
Out[486]:
DatetimeIndex(['2015-03-29 01:59:59.999999999+01:00',
              '2015-03-29 03:30:00+02:00',
              '2015-03-29 04:30:00+02:00'],
       dtype='datetime64[ns, Europe/Warsaw]', freq=None)

In [487]: dti.tz_localize("Europe/Warsaw", nonexistent=pd.Timedelta(1, unit="H"))
Out[487]:
DatetimeIndex(['2015-03-29 03:30:00+02:00', '2015-03-29 03:30:00+02:00',
              '2015-03-29 04:30:00+02:00'],
       dtype='datetime64[ns, Europe/Warsaw]', freq=None)

In [488]: dti.tz_localize("Europe/Warsaw", nonexistent="NaT")
Out[488]:
DatetimeIndex(['NaT', '2015-03-29 03:30:00+02:00',
              '2015-03-29 04:30:00+02:00'],
       dtype='datetime64[ns, Europe/Warsaw]', freq=None)

Time zone series operations

A Series with time zone naive values is represented with a dtype of datetime64[ns].

In [489]: s_naive = pd.Series(pd.date_range("20130101", periods=3))

In [490]: s_naive
Out[490]:
 0   2013-01-01
1   2013-01-02
2   2013-01-03
dtype: datetime64[ns]

A Series with a time zone aware values is represented with a dtype of datetime64[ns, tz] where tz is the time zone

In [491]: s_aware = pd.Series(pd.date_range("20130101", periods=3, tz="US/Eastern"))

In [492]: s_aware
Out[492]:
 0   2013-01-01 00:00:00-05:00
(continues on next page)
Both of these `Series` time zone information can be manipulated via the `.dt` accessor, see the `dt accessor section`.

For example, to localize and convert a naive stamp to time zone aware.

```python
In [493]: s_naive.dt.tz_localize("UTC").dt.tz_convert("US/Eastern")
Out[493]:
0  2012-12-31 19:00:00-05:00
1  2013-01-01 19:00:00-05:00
2  2013-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

Time zone information can also be manipulated using the `astype` method. This method can convert between different time zone-aware dtypes.

```python
# convert to a new time zone
In [494]: s_aware.astype("datetime64[ns, CET"]
Out[494]:
0  2013-01-01 06:00:00+01:00
1  2013-01-02 06:00:00+01:00
2  2013-01-03 06:00:00+01:00
dtype: datetime64[ns, CET]
```

**Note:** Using `Series.to_numpy()` on a `Series`, returns a NumPy array of the data. NumPy does not currently support time zones (even though it is printing in the local time zone!), therefore an object array of Timestamps is returned for time zone aware data:

```python
In [495]: s_naive.to_numpy()
Out[495]:
array([Timestamp('2013-01-01 00:00:00-05:00', tz='US/Eastern'),
       Timestamp('2013-01-02 00:00:00-05:00', tz='US/Eastern'),
       Timestamp('2013-01-03 00:00:00-05:00', tz='US/Eastern')],
dtype='object')
```

By converting to an object array of Timestamps, it preserves the time zone information. For example, when converting back to a Series:

```python
In [496]: pd.Series(s_aware.to_numpy())
Out[496]:
0  2013-01-01 00:00:00-05:00
1  2013-01-02 00:00:00-05:00
2  2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

However, if you want an actual NumPy `datetime64[ns]` array (with the values converted to UTC) instead of an array...
of objects, you can specify the dtype argument:

```python
In [498]: s_aware.to_numpy(dtype="datetime64[ns]"
Out[498]:
array(['2013-01-01T05:00:00.000000000', '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

## 2.21 Time deltas

Timedeltas are differences in times, expressed in difference units, e.g. days, hours, minutes, seconds. They can be both positive and negative.

Timedelta is a subclass of datetime.timedelta, and behaves in a similar manner, but allows compatibility with np.timedelta64 types as well as a host of custom representation, parsing, and attributes.

### 2.21.1 Parsing

You can construct a Timedelta scalar through various arguments, including ISO 8601 Duration strings.

```python
In [1]: import datetime

# strings
In [2]: pd.Timedelta("1 days")
Out[2]: Timedelta('1 days 00:00:00')

In [3]: pd.Timedelta("1 days 00:00:00")
Out[3]: Timedelta('1 days 00:00:00')

In [4]: pd.Timedelta("1 days 2 hours")
Out[4]: Timedelta('1 days 02:00:00')

In [5]: pd.Timedelta("-1 days 2 min 3us")
Out[5]: Timedelta('-2 days +23:57:59.999997')

# like datetime.timedelta
# note: these MUST be specified as keyword arguments
In [6]: pd.Timedelta(days=1, seconds=1)
Out[6]: Timedelta('1 days 00:00:01')

# integers with a unit
In [7]: pd.Timedelta(1, unit="d")
Out[7]: Timedelta('1 days 00:00:00')

# from a datetime.timedelta/np.timedelta64
In [8]: pd.Timedelta(datetime.timedelta(days=1, seconds=1))
Out[8]: Timedelta('1 days 00:00:01')

In [9]: pd.Timedelta(np.timedelta64(1, "ms"))
Out[9]: Timedelta('0 days 00:00:00.001000')
```

(continues on next page)
negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [10]: pd.Timedelta("-1us")
Out[10]: Timedelta('-1 days +23:59:59.999999')

# a NaT
In [11]: pd.Timedelta("nan")
Out[11]: NaT
In [12]: pd.Timedelta("nat")
Out[12]: NaT

# ISO 8601 Duration strings
In [13]: pd.Timedelta("P0DT0H1M0S")
Out[13]: Timedelta('0 days 00:01:00')
In [14]: pd.Timedelta("P0DT0H0M0.000000123S")
Out[14]: Timedelta('0 days 00:00:00.000000123')

DateOffsets (Day, Hour, Minute, Second, Milli, Micro, Nano) can also be used in construction.
In [15]: pd.Timedelta(pd.offsets.Second(2))
Out[15]: Timedelta('0 days 00:00:02')

Further, operations among the scalars yield another scalar Timedelta.
In [16]: pd.Timedelta(pd.offsets.Day(2)) + pd.Timedelta(pd.offsets.Second(2)) + pd.˓→Timedelta(
 .....:   "00:00:02.000123"
 .....: )
 .....:
Out[16]: Timedelta('2 days 00:00:02.000123')

to_timedelta

Using the top-level pd.to_timedelta, you can convert a scalar, array, list, or Series from a recognized timedelta format / value into a Timedelta type. It will construct Series if the input is a Series, a scalar if the input is scalar-like, otherwise it will output a TimedeltaIndex.

You can parse a single string to a Timedelta:
In [17]: pd.to_timedelta("1 days 06:05:01.00003")
Out[17]: Timedelta('1 days 06:05:01.00003')
In [18]: pd.to_timedelta("15.5us")
Out[18]: Timedelta('0 days 00:00:00.000015500')

or a list/array of strings:
In [19]: pd.to_timedelta(["1 days 06:05:01.00003", "15.5us", "nan"])
Out[19]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015500', NaT],˓→dtype='timedelta64[ns]', freq=None)
The unit keyword argument specifies the unit of the Timedelta if the input is numeric:

```python
In [20]: pd.to_timedelta(np.arange(5), unit="s")
Out[20]:
TimedeltaIndex(['0 days 00:00:00', '0 days 00:00:01', '0 days 00:00:02',
 '0 days 00:00:03', '0 days 00:00:04'],
dtype='timedelta64[ns]', freq=None)
```

```python
In [21]: pd.to_timedelta(np.arange(5), unit="d")
Out[21]:
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
˓→'timedelta64[ns]', freq=None)
```

**Warning:** If a string or array of strings is passed as an input then the `unit` keyword argument will be ignored. If a string without units is passed then the default unit of nanoseconds is assumed.

### Timedelta limitations

pandas represents Timedeltas in nanosecond resolution using 64 bit integers. As such, the 64 bit integer limits determine the Timedelta limits.

```python
In [22]: pd.Timedelta.min
Out[22]: Timedelta('-106752 days +00:12:43.145224193')
```

```python
In [23]: pd.Timedelta.max
Out[23]: Timedelta('106751 days 23:47:16.854775807')
```

#### 2.2.1.2 Operations

You can operate on Series/DataFrames and construct timedelta64[ns] Series through subtraction operations on datetime64[ns] Series, or Timestamps.

```python
In [24]: s = pd.Series(pd.date_range("2012-1-1", periods=3, freq="D"))
In [25]: td = pd.Series([pd.Timedelta(days=i) for i in range(3)])
In [26]: df = pd.DataFrame({'A': s, 'B': td})
In [27]: df
Out[27]:
   A         B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days
```

```python
In [28]: df["C"] = df["A"] + df["B"]
In [29]: df
Out[29]:
   A         B       C
0 2012-01-01 0 days 2012-01-01
```

(continues on next page)
In [30]: df.dtypes
Out[30]:
A   datetime64[ns]
B   timedelta64[ns]
C   datetime64[ns]
dtype: object

In [31]: s - s.max()
Out[31]:
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [32]: s - pd.datetime(2011, 1, 1, 3, 5)
Out[32]:
0  364 days 20:55:00
1  365 days 20:55:00
2  366 days 20:55:00
dtype: timedelta64[ns]

In [33]: s + pd.offsets.Minute(5)
Out[33]:
0  2012-01-01 00:05:00
1  2012-01-02 00:05:00
2  2012-01-03 00:05:00
dtype: datetime64[ns]

Operations with scalars from a timedelta64[ns] series:

In [36]: y = s - s[0]
In [37]: y
Out[37]:
0    0 days
(continues on next page)
Series of timedeltas with NaT values are supported:

```python
In [38]: y = s - s.shift()
In [39]: y
Out[39]:
0   NaT
1   1 days
2   1 days
dtype: timedelta64[ns]
```

Elements can be set to NaT using np.nan analogously to datetimes:

```python
In [40]: y[1] = np.nan
In [41]: y
Out[41]:
0   NaT
1   NaT
2   1 days
dtype: timedelta64[ns]
```

Operands can also appear in a reversed order (a singular object operated with a Series):

```python
In [42]: s.max() - s
Out[42]:
0   2 days
1   1 days
2   0 days
dtype: timedelta64[ns]
```

```python
In [43]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[43]:
0   -365 days +03:05:00
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]
```

```python
In [44]: datetime.timedelta(minutes=5) + s
Out[44]:
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

`min`, `max` and the corresponding `idxmin`, `idxmax` operations are supported on frames:

```python
In [45]: A = s - pd.Timestamp("20120101") - pd.Timedelta("00:05:05")
```

(continues on next page)
In [46]: B = s - pd.Series(pd.date_range("2012-1-2", periods=3, freq="D"))

In [47]: df = pd.DataFrame({'A': A, 'B': B})

In [48]: df
Out[48]:
   A          B
0 -1 days +23:54:55 -1 days
1 0 days 23:54:55 -1 days
2 1 days 23:54:55 -1 days

In [49]: df.min()
Out[49]:
   A          B
0 -1 days +23:54:55
1 -1 days +00:00:00
dtype: timedelta64[ns]

In [50]: df.min(axis=1)
Out[50]:
0 -1 days
1 -1 days
2 -1 days
dtype: timedelta64[ns]

In [51]: df.idxmin()
Out[51]:
   A   B
0   0   0
dtype: int64

In [52]: df.idxmax()
Out[52]:
   A   B
0   2   0
dtype: int64

min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

In [53]: df.min().max()
Out[53]: Timedelta('-1 days +23:54:55')

In [54]: df.min(axis=1).min()
Out[54]: Timedelta('-1 days +00:00:00')

In [55]: df.min().idxmax()
Out[55]: 'A'

In [56]: df.min(axis=1).idxmin()
Out[56]: 0

You can fillna on timedeltas, passing a timedelta to get a particular value.
In [57]: y.fillna(pd.Timedelta(0))
Out[57]:
0  0 days
1  0 days
2  1 days
dtype: timedelta64[ns]

In [58]: y.fillna(pd.Timedelta(10, unit="s"))
Out[58]:
0  0 days 00:00:10
1  0 days 00:00:10
2  1 days 00:00:00
dtype: timedelta64[ns]

In [59]: y.fillna(pd.Timedelta("-1 days, 00:00:05"))
Out[59]:
0  -1 days +00:00:05
1  -1 days +00:00:05
2   1 days 00:00:00
dtype: timedelta64[ns]

You can also negate, multiply and use abs on Timedeltas:

In [60]: td1 = pd.Timedelta("-1 days 2 hours 3 seconds")
In [61]: td1
Out[61]: Timedelta(-2 days +21:59:57)

In [62]: -1 * td1
Out[62]: Timedelta('1 days 02:00:03')

In [63]: -td1
Out[63]: Timedelta('1 days 02:00:03')

In [64]: abs(td1)
Out[64]: Timedelta('1 days 02:00:03')

2.21.3 Reductions

Numeric reduction operation for timedelta64[ns] will return Timedelta objects. As usual NaT are skipped during evaluation.

In [65]: y2 = pd.Series(
    ....:     pd.to_timedelta(["-1 days +00:00:05", "nat", "-1 days +00:00:05", "1 days"])
    ....: )
    ....:

In [66]: y2
Out[66]:
0  -1 days +00:00:05
1       NaT
2  -1 days +00:00:05

(continues on next page)
3  1 days 00:00:00
dtype: timedelta64[ns]

In [67]: y2.mean()
Out[67]: Timedelta('-1 days +16:00:03.333333334')

In [68]: y2.median()
Out[68]: Timedelta('-1 days +00:00:05')

In [69]: y2.quantile(0.1)
Out[69]: Timedelta('-1 days +00:00:05')

In [70]: y2.sum()
Out[70]: Timedelta('-1 days +00:00:10')

2.21.4 Frequency conversion

Timedelta Series, TimedeltaIndex, and Timedelta scalars can be converted to other 'frequencies' by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield Series and propagate NaT -> nan. Note that division by the NumPy scalar is true division, while astyping is equivalent of floor division.

In [71]: december = pd.Series(pd.date_range("20121201", periods=4))

In [72]: january = pd.Series(pd.date_range("20130101", periods=4))

In [73]: td = january - december

In [74]: td[2] += datetime.timedelta(minutes=5, seconds=3)

In [75]: td[3] = np.nan

In [76]: td
Out[76]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 NaT
dtype: timedelta64[ns]

# to days
In [77]: td / np.timedelta64(1, "D")
Out[77]:
0 31.000000
1 31.000000
2 31.005000
3 NaN
dtype: float64

In [78]: td.astype("timedelta64[D]")
Out[78]:
0 31.0

(continues on next page)
Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series yields another timedelta64[ns] dtypes Series.

Rounded division (floor-division) of a timedelta64[ns] Series by a scalar Timedelta gives a series of integers.
In [84]: td // pd.Timedelta(days=3, hours=4)
Out[84]:
0  31.0
1  31.0
2  31.0
3  NaN
dtype: float64

In [85]: pd.Timedelta(days=3, hours=4) // td
Out[85]:
0  0.0
1  0.0
2  0.0
3  NaN
dtype: float64

The mod (%) and divmod operations are defined for Timedelta when operating with another timedelta-like or with a numeric argument.

In [86]: pd.Timedelta(hours=37) % datetime.timedelta(hours=2)
Out[86]: Timedelta('0 days 01:00:00')

# divmod against a timedelta-like returns a pair (int, Timedelta)
In [87]: divmod(datetime.timedelta(hours=2), pd.Timedelta(minutes=11))
Out[87]: (10, Timedelta('0 days 00:10:00'))

# divmod against a numeric returns a pair (Timedelta, Timedelta)
In [88]: divmod(pd.Timedelta(hours=25), 86400000000000)
Out[88]: (Timedelta('0 days 00:00:00.000000001'), Timedelta('0 days 01:00:00'))

2.21.5 Attributes

You can access various components of the Timedelta or TimedeltaIndex directly using the attributes days, seconds, microseconds, nanoseconds. These are identical to the values returned by datetime.timedelta, in that, for example, the .seconds attribute represents the number of seconds >= 0 and < 1 day. These are signed according to whether the Timedelta is signed.

These operations can also be directly accessed via the .dt property of the Series as well.

Note: Note that the attributes are NOT the displayed values of the Timedelta. Use .components to retrieve the displayed values.
In [90]: td.dt.seconds
Out[90]:
0     0.0
1     0.0
2  303.0
3     NaN
dtype: float64

You can access the value of the fields for a scalar Timedelta directly.

In [91]: tds = pd.Timedelta("31 days 5 min 3 sec")
In [92]: tds.days
Out[92]: 31
In [93]: tds.seconds
Out[93]: 303
In [94]: (-tds).seconds
Out[94]: 86097

You can use the .components property to access a reduced form of the timedelta. This returns a DataFrame indexed similarly to the Series. These are the displayed values of the Timedelta.

In [95]: td.dt.components
Out[95]:
                     days    hours    minutes   seconds  milliseconds  microseconds nanosecond
0   31.000000    0.000000    0.000000    0.000000      0.000000      0.000000      0.000000
1   31.000000    0.000000    0.000000    0.000000      0.000000      0.000000      0.000000
2   31.000000    0.000000    0.083333    3.000000      0.000000      0.000000      0.000000
3     NaN       NaN       NaN       NaN         NaN         NaN         NaN

In [96]: td.dt.components.seconds
Out[96]:
0     0.0
1     0.0
2     3.0
3     NaN
Name: seconds, dtype: float64

You can convert a Timedelta to an ISO 8601 Duration string with the .isoformat method.

In [97]: pd.Timedelta(  
...:     days=6, minutes=50, seconds=3, milliseconds=10, microseconds=10,  
...:     nanoseconds=12  
...:     ).isoformat()  
...:     )
Out[97]: 'P6DT0H50M3.010010012S'
2.21.6 TimedeltaIndex

To generate an index with time delta, you can use either the `TimedeltaIndex` or the `timedelta_range()` constructor.

Using `TimedeltaIndex` you can pass string-like, `Timedelta`, `timedelta`, or `np.timedelta64` objects. Passing `np.nan/pd.NaT/nat` will represent missing values.

```python
In [98]: pd.TimedeltaIndex(
   ....:     ["1 days", "1 days, 00:00:05", np.timedelta64(2, "D"),
   ....:      datetime.timedelta(days=2, seconds=2),
   ....:   ]
   ....: )
Out[98]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
   '2 days 00:00:02'], dtype='timedelta64[ns]', freq=None)
```

The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation:

```python
In [99]: pd.TimedeltaIndex(['0 days', '10 days', '20 days'], freq="infer")
Out[99]: TimedeltaIndex(['0 days', '10 days', '20 days'], dtype='timedelta64[ns]', freq='10D')
```

Generating ranges of time deltas

Similar to `date_range()`, you can construct regular ranges of a `TimedeltaIndex` using `timedelta_range()`. The default frequency for `timedelta_range` is calendar day:

```python
In [100]: pd.timedelta_range(start="1 days", periods=5)
Out[100]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq='D')
```

Various combinations of start, end, and periods can be used with `timedelta_range`:

```python
In [101]: pd.timedelta_range(start="1 days", end="5 days")
Out[101]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq='D')
```

```python
In [102]: pd.timedelta_range(end="10 days", periods=4)
Out[102]: TimedeltaIndex(['7 days', '8 days', '9 days', '10 days'], dtype='timedelta64[ns]', freq='D')
```

The `freq` parameter can passed a variety of `frequency aliases`:

```python
In [103]: pd.timedelta_range(start="1 days", end="2 days", freq="30T")
Out[103]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
   '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
   '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00'],
   (continues on next page)
2.21. Time deltas

Specifying start, end, and periods will generate a range of evenly spaced timedeltas from start to end inclusively, with periods number of elements in the resulting TimedeltaIndex:

```python
In [105]: pd.timedelta_range("0 days", "4 days", periods=5)
Out[105]:
TimedeltaIndex(['0 days 00:00:00', '1 days 00:00:00', '2 days 00:00:00', '3 days 00:00:00', '4 days 00:00:00'], dtype='timedelta64[ns]', freq=None)
```

Using the TimedeltaIndex

Similarly to other of the datetime-like indices, DatetimeIndex and PeriodIndex, you can use TimedeltaIndex as the index of pandas objects.

```python
In [107]: s = pd.Series(
            np.arange(100),
            index=pd.timedelta_range("1 days", periods=100, freq="h"),
       )

In [108]: s
Out[108]:
1 days 00:00:00    0
```

(continues on next page)
Selections work similarly, with coercion on string-likes and slices:

```python
In [109]: s["1 day":"2 day"]
Out[109]:
1 days 00:00:00 0
1 days 01:00:00 1
1 days 02:00:00 2
1 days 03:00:00 3
1 days 04:00:00 4
..  
2 days 19:00:00 43
2 days 20:00:00 44
2 days 21:00:00 45
2 days 22:00:00 46
2 days 23:00:00 47
Freq: H, Length: 48, dtype: int64
```

```python
In [110]: s["1 day 01:00:00"]
Out[110]: 1
```

```python
In [111]: s[pd.Timedelta("1 day 1h")]
Out[111]: 1
```

Furthermore you can use partial string selection and the range will be inferred:

```python
In [112]: s["1 day":"1 day 5 hours"]
Out[112]:
1 days 00:00:00 0
1 days 01:00:00 1
1 days 02:00:00 2
1 days 03:00:00 3
1 days 04:00:00 4
1 days 05:00:00 5
Freq: H, dtype: int64
```
Operations

Finally, the combination of TimedeltaIndex with DatetimeIndex allow certain combination operations that are NaT preserving:

```python
In [113]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [114]: tdi.to_list()
Out[114]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [115]: dti = pd.date_range('20130101', periods=3)

In [116]: dti.to_list()
Out[116]: [Timestamp('2013-01-01 00:00:00', freq='D'),
         Timestamp('2013-01-02 00:00:00', freq='D'),
         Timestamp('2013-01-03 00:00:00', freq='D')]

In [117]: (dti + tdi).to_list()
Out[117]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [118]: (dti - tdi).to_list()
Out[118]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

Conversions

Similarly to frequency conversion on a Series above, you can convert these indices to yield another Index.

```python
In [119]: tdi / np.timedelta64(1, 's')
Out[119]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

In [120]: tdi.astype('timedelta64[s]')
Out[120]: Float64Index([86400.0, nan, 172800.0], dtype='float64')
```

Scalars type ops work as well. These can potentially return a different type of index.

```python
# adding or timedelta and date -> datelike
In [121]: tdi + pd.Timestamp('20130101')
Out[121]: DatetimeIndex(['2013-01-02', 'NaT', '2013-01-03'], dtype='datetime64[ns]',
                      freq=None)

# subtraction of a date and a timedelta -> datelike
# note that trying to subtract a date from a Timedelta will raise an exception
In [122]: (pd.Timestamp('20130101') - tdi).to_list()
Out[122]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')]

# timedelta + timedelta -> timedelta
In [123]: tdi + pd.Timedelta('10 days')
Out[123]: TimedeltaIndex(['10 days'], freq=None)

# division can result in a Timedelta if the divisor is an integer
In [124]: tdi / 2
Out[124]: TimedeltaIndex(['5 days 00:00:00', 'NaT', '5 days 00:00:00'], dtype=
                       'timedelta64[ms]', freq=None)
```

(continues on next page)
2.21.7 Resampling

Similar to timeseries resampling, we can resample with a TimedeltaIndex.

```python
In [126]: s.resample("D").mean()
```

```
Out[126]:
1 days  11.5
2 days  35.5
3 days  59.5
4 days  83.5
5 days  97.5
Freq: D, dtype: float64
```

2.22 Options and settings

2.22.1 Overview

pandas has an options API configure and customize global behavior related to DataFrame display, data behavior and more.

Options have a full “dotted-style”, case-insensitive name (e.g. display.max_rows). You can get/set options directly as attributes of the top-level options attribute:

```python
In [1]: import pandas as pd
In [2]: pd.options.display.max_rows
Out[2]:
In [3]: pd.options.display.max_rows = 999
```

The API is composed of 5 relevant functions, available directly from the pandas namespace:

- `get_option() / set_option()` - get/set the value of a single option.
- `reset_option()` - reset one or more options to their default value.
- `describe_option()` - print the descriptions of one or more options.
- `option_context()` - execute a codeblock with a set of options that revert to prior settings after execution.

**Note:** Developers can check out pandas/core/config_init.py for more information.
All of the functions above accept a regexp pattern (`re.search` style) as an argument, to match an unambiguous substring:

```
In [5]: pd.get_option("display.chop_threshold")

In [6]: pd.set_option("display.chop_threshold", 2)

In [7]: pd.get_option("display.chop_threshold")
Out[7]: 2

In [8]: pd.set_option("chop", 4)

In [9]: pd.get_option("display.chop_threshold")
Out[9]: 4
```

The following will not work because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```
In [10]: pd.get_option("max")

---------------------------------------------------------------------------
OptionError                                 Traceback (most recent call last)
Input In [10], in <cell line: 1>()
    1 pd.get_option("max")
--> 2 pd.get_option("max")
File /pandas/pandas/_config/config.py:256, in CallableDynamicDoc.__call__(self, *args, **kwds)
    255     def __call__(self, *args, **kwds):
--> 256         return self.__func__(*args, **kwds)
File /pandas/pandas/_config/config.py:128, in _get_option(pat, silent)
    127     # walk the nested dict
    128     root, k = _get_root(key)
File /pandas/pandas/_config/config.py:116, in _get_single_key(pat, silent)
    114     raise OptionError(f"No such keys(s): {repr(pat)}")
    115 if len(keys) > 1:
--> 116     raise OptionError("Pattern matched multiple keys")
    117     key = keys[0]
    119 if not silent:

OptionError: 'Pattern matched multiple keys'
```

**Warning**: Using this form of shorthand may cause your code to break if new options with similar names are added in future versions.
2.22.2 Available options

You can get a list of available options and their descriptions with `describe_option()`. When called with no argument `describe_option()` will print out the descriptions for all available options.

```
In [11]: pd.describe_option()
```

- **compute.use_bottleneck**: bool
  Use the bottleneck library to accelerate if it is installed, the default is True
  Valid values: False, True
  [default: True] [currently: True]

- **compute.use_numba**: bool
  Use the numba engine option for select operations if it is installed, the default is False
  Valid values: False, True
  [default: False] [currently: False]

- **compute.use_numexpr**: bool
  Use the numexpr library to accelerate computation if it is installed, the default is True
  Valid values: False, True
  [default: True] [currently: True]

- **display.chop_threshold**: float or None
  If set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends.
  [default: None] [currently: None]

- **display.colheader_justify**: 'left'/'right'
  Controls the justification of column headers. used by DataFrameFormatter.
  [default: right] [currently: right]

- **display.column_space**: No description available.
  [default: 12] [currently: 12]

- **display.date_dayfirst**: boolean
  When True, prints and parses dates with the day first, eg 20/01/2005
  [default: False] [currently: False]

- **display.date_yearfirst**: boolean
  When True, prints and parses dates with the year first, eg 2005/01/20
  [default: False] [currently: False]

- **display.encoding**: str/unicode
  Defaults to the detected encoding of the console.
  Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console.
  [default: utf-8] [currently: utf8]

- **display.expand_frame_repr**: boolean
  Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, `max_columns` is still respected, but the output will wrap-around across multiple "pages" if its width exceeds `display.width`.
  [default: True] [currently: True]

- **display.float_format**: callable
  The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter.
  See formats.format.EngFormatter for an example.
  [default: None] [currently: None]

- **display.html.border**: int

(continues on next page)
A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr.

- `display.html.table_schema : boolean`
  - Whether to publish a Table Schema representation for frontends that support it.
  - (default: False)
  - [default: False] [currently: False]

- `display.html.use_mathjax : boolean`
  - When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol.
  - (default: True)
  - [default: True] [currently: True]

- `display.large_repr : 'truncate'/'info'`
  - For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas).
  - [default: truncate] [currently: truncate]

- `display.latex.escape : bool`
  - This specifies if the to_latex method of a Dataframe uses escapes special characters.
  - Valid values: False,True
  - [default: True] [currently: True]

- `display.latex.longtable : bool`
  - This specifies if the to_latex method of a Dataframe uses the longtable format.
  - Valid values: False,True
  - [default: False] [currently: False]

- `display.latex.multicolumn : bool`
  - This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns.
  - Valid values: False,True
  - [default: True] [currently: True]

- `display.latex.multicolumn_format : bool`
  - This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns.
  - Valid values: False,True
  - [default: l] [currently: l]

- `display.latex.multirow : bool`
  - This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows.
  - Valid values: False,True
  - [default: False] [currently: False]

- `display.latex.repr : boolean`
  - Whether to produce a latex DataFrame representation for jupyter environments that support it.
  - (default: False)
  - [default: False] [currently: False]

- `display.max_categories : int`
  - This sets the maximum number of categories pandas should output when printing out a `Categorical` or a Series of dtype "category".
  - [default: 8] [currently: 8]
display.max_columns : int
If max_cols is exceeded, switch to truncate view. Depending on 'large_repr', objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and 'large_repr' equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection.
[default: 0] [currently: 0]

display.max_colwidth : int or None
The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "..." placeholder is embedded in the output. A 'None' value means unlimited.
[default: 50] [currently: 50]

display.max_dir_items : int
The number of items that will be added to `dir(...)`. 'None' value means unlimited. Because dir is cached, changing this option will not immediately affect already existing dataframes until a column is deleted or added.

This is for instance used to suggest columns from a dataframe to tab completion.
[default: 100] [currently: 100]

display.max_info_columns : int
max_info_columns is used in DataFrame.info method to decide if per column information will be printed.
[default: 100] [currently: 100]

display.max_info_rows : int or None
df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified.
[default: 1690785] [currently: 1690785]

display.max_rows : int
If max_rows is exceeded, switch to truncate view. Depending on 'large_repr', objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and 'large_repr' equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection.
[default: 60] [currently: 60]

display.max_seq_items : int or None
When pretty-printing a long sequence, no more than `max_seq_items` will be printed. If items are omitted, they will be denoted by the addition of "..." to the resulting string.
If set to None, the number of items to be printed is unlimited.
[default: 100] [currently: 100]

display.memory_usage : bool, string or None
This specifies if the memory usage of a DataFrame should be displayed when
df.info() is called. Valid values True, False, 'deep'
[default: True] [currently: True]

display.min_rows : int
The numbers of rows to show in a truncated view (when `max_rows` is
exceeded). Ignored when `max_rows` is set to None or 0. When set to
None, follows the value of `max_rows`.
[default: 10] [currently: 10]

display.multi_sparse : boolean
"sparsify" MultiIndex display (don't display repeated
elements in outer levels within groups)
[default: True] [currently: True]

display.notebook_repr_html : boolean
When True, IPython notebook will use html representation for
pandas objects (if it is available).
[default: True] [currently: True]

display.pprint_nest_depth : int
Controls the number of nested levels to process when pretty-printing
[default: 3] [currently: 3]

display.precision : int
Floating point output precision in terms of number of places after the
decimal, for regular formatting as well as scientific notation. Similar
to `precision` in :meth:`numpy.set_printoptions`.
[default: 6] [currently: 6]

display.show_dimensions : boolean or 'truncate'
Whether to print out dimensions at the end of DataFrame repr.
If 'truncate' is specified, only print out the dimensions if the
frame is truncated (e.g. not display all rows and/or columns)
[default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide : boolean
Whether to use the Unicode East Asian Width to calculate the display text
width.
Enabling this may affect to the performance (default: False)
[default: False] [currently: False]

display.unicode.east_asian_width : boolean
Whether to use the Unicode East Asian Width to calculate the display text
width.
Enabling this may affect to the performance (default: False)
[default: False] [currently: False]

display.width : int
Width of the display in characters. In case python/IPython is running in
a terminal this can be set to None and pandas will correctly auto-detect
the width.
Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a
terminal and hence it is not possible to correctly detect the width.
[default: 80] [currently: 80]

io.excel.ods.reader : string
The default Excel reader engine for 'ods' files. Available options:
auto, odf.
io.excel.ods.writer : string
  The default Excel writer engine for 'ods' files. Available options:
  auto, odf.
  [default: auto] [currently: auto]

io.excel.xls.reader : string
  The default Excel reader engine for 'xls' files. Available options:
  auto, xlrd.
  [default: auto] [currently: auto]

io.excel.xls.writer : string
  The default Excel writer engine for 'xls' files. Available options:
  auto, xlwt.
  [default: auto] [currently: auto]

io.excel.xlsb.reader : string
  The default Excel reader engine for 'xlsb' files. Available options:
  auto, pyxlsb.
  [default: auto] [currently: auto]

io.excel.xlsx.reader : string
  The default Excel reader engine for 'xlsx' files. Available options:
  auto, xld, openpyxl.
  [default: auto] [currently: auto]

io.excel.xlsx.writer : string
  The default Excel writer engine for 'xlsx' files. Available options:
  auto, openpyxl, xlsxwriter.
  [default: auto] [currently: auto]

io.hdf.default_format : format
  default format writing format, if None, then
  put will default to 'fixed' and append will default to 'table'
  [default: None] [currently: None]

io.hdf.dropna_table : boolean
  drop ALL nan rows when appending to a table
  [default: False] [currently: False]

io.parquet.engine : string
  The default parquet reader/writer engine. Available options:
  'auto', 'pyarrow', 'fastparquet', the default is 'auto'
  [default: auto] [currently: auto]

io.sql.engine : string
  The default sql reader/writer engine. Available options:
  'auto', 'sqlalchemy', the default is 'auto'
  [default: auto] [currently: auto]

mode.chained_assignment : string
  Raise an exception, warn, or no action if trying to use chained assignment,
  The default is warn
### Options and settings

<table>
<thead>
<tr>
<th>Key</th>
<th>Default</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode.data_manager</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td>mode.sim_interactive</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>mode.string_storage</td>
<td>python</td>
<td>python</td>
</tr>
<tr>
<td>mode.use_inf_as_na</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>mode.use_inf_as_null</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>plotting.backend</td>
<td>matplotlib</td>
<td>matplotlib</td>
</tr>
<tr>
<td>plotting.matplotlib.register_converters</td>
<td>auto</td>
<td>auto</td>
</tr>
<tr>
<td>styler.format.decimal</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>styler.format.escape</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>styler.format.formatter</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>styler.format.na_rep</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>styler.format.precision</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>styler.format.thousands</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>styler.html.mathjax</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

(continues on next page)
will not be used in Jupyter Notebook.
[default: True] [currently: True]

styler.latex.environment : str
The environment to replace `\begin{table}`. If "longtable" is used results in a specific longtable environment format.
[default: None] [currently: None]

styler.latex.hrules : bool
Whether to add horizontal rules on top and bottom and below the headers.
[default: False] [currently: False]

styler.latex.multicol_align : {"r", "c", "l", "naive-l", "naive-r"}
The specifier for horizontal alignment of sparsified LaTeX multicolumns. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. "\r" will draw a rule on the left side of right aligned merged cells.
[default: r] [currently: r]

styler.latex.mutirow_align : {"c", "t", "b"}
The specifier for vertical alignment of sparsified LaTeX multirows.
[default: c] [currently: c]

styler.render.encoding : str
The encoding used for output HTML and LaTeX files.
[default: utf-8] [currently: utf-8]

styler.render.max_columns : int, optional
The maximum number of columns that will be rendered. May still be reduced to satisfy `max_elements
[default: None] [currently: None]

styler.render.max_elements : int
The maximum number of data-cell (<td>) elements that will be rendered before trimming will occur over columns, rows or both if needed.
[default: 262144] [currently: 262144]

styler.render.max_rows : int, optional
The maximum number of rows that will be rendered. May still be reduced to satisfy `max_elements`, which takes precedence.
[default: None] [currently: None]

styler.render.repr : str
Determine which output to use in Jupyter Notebook in {"html", "latex"}.
[default: html] [currently: html]

styler.sparse.columns : bool
Whether to sparsify the display of hierarchical columns. Setting to False will display each explicit level element in a hierarchical key for each column.
[default: True] [currently: True]

styler.sparse.index : bool
Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row.
[default: True] [currently: True]
2.22.3 Getting and setting options

As described above, \texttt{get_option()} and \texttt{set_option()} are available from the pandas namespace. To change an option, call \texttt{set_option('option regex', new_value)}.

In [12]: \texttt{pd.get_option('mode.sim_interactive')}
Out[12]: False

In [13]: \texttt{pd.set_option('mode.sim_interactive', True)}

In [14]: \texttt{pd.get_option('mode.sim_interactive')}
Out[14]: True

\textbf{Note:} The option 'mode.sim_interactive' is mostly used for debugging purposes.

You can use \texttt{reset_option()} to revert to a setting’s default value

In [15]: \texttt{pd.get_option('display.max_rows')}
Out[15]: 60

In [16]: \texttt{pd.set_option('display.max_rows', 999)}

In [17]: \texttt{pd.get_option('display.max_rows')}
Out[17]: 999

In [18]: \texttt{pd.reset_option('display.max_rows')}

In [19]: \texttt{pd.get_option('display.max_rows')}
Out[19]: 60

It’s also possible to reset multiple options at once (using a regex):

In [20]: \texttt{pd.reset_option('^display')}

\texttt{option_context()} context manager has been exposed through the top-level API, allowing you to execute code with given option values. Option values are restored automatically when you exit the with block:

In [21]: \texttt{with pd.option_context('display.max_rows', 10, 'display.max_columns', 5):}
   ....: \texttt{print(pd.get_option('display.max_rows'))}
   ....: \texttt{print(pd.get_option('display.max_columns'))}
   ....:
10
5

In [22]: \texttt{print(pd.get_option('display.max_rows'))}
Out[22]: 60

In [23]: \texttt{print(pd.get_option('display.max_columns'))}
Out[23]: 0
2.22.4 Setting startup options in Python/IPython environment

Using startup scripts for the Python/IPython environment to import pandas and set options makes working with pandas more efficient. To do this, create a .py or .ipy script in the startup directory of the desired profile. An example where the startup folder is in a default IPython profile can be found at:

```bash
$IPYTHONDIR/profile_default/startup
```

More information can be found in the IPython documentation. An example startup script for pandas is displayed below:

```python
import pandas as pd

pd.set_option("display.max_rows", 999)
pd.set_option("display.precision", 5)
```

2.22.5 Frequently used options

The following is a demonstrates the more frequently used display options.

display.max_rows and display.max_columns sets the maximum number of rows and columns displayed when a frame is pretty-printed. Truncated lines are replaced by an ellipsis.

```
In [24]: df = pd.DataFrame(np.random.randn(7, 2))
In [25]: pd.set_option("display.max_rows", 7)
In [26]: df
Out[26]:
     0       1
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569
5 -0.494929  1.071804
6  0.721555 -0.706771
In [27]: pd.set_option("display.max_rows", 5)
In [28]: df
Out[28]:
     0       1
0  0.469112 -0.282863
1 -1.509059 -1.135632
...       ...       ...
5 -0.494929  1.071804
6  0.721555 -0.706771
[7 rows x 2 columns]
```

Once the display.max_rows is exceeded, the display.min_rows options determines how many rows are shown in the truncated repr.
In [30]: pd.set_option("display.max_rows", 8)
In [31]: pd.set_option("display.min_rows", 4)

# below max_rows -> all rows shown
In [32]: df = pd.DataFrame(np.random.randn(7, 2))
In [33]: df
Out[33]:
   0   1
0 -1.039575  0.271860
1 -0.424972  0.567020
2  0.276232 -1.087401
3 -0.673690  0.113648
4 -1.478427  0.524988
5  0.404705  0.577046
6 -1.715002 -1.039268

# above max_rows -> only min_rows (4) rows shown
In [34]: df = pd.DataFrame(np.random.randn(9, 2))
In [35]: df
Out[35]:
   0   1
0 -0.370647 -1.157892
1 -1.344312  0.844885
... ... ...
7  0.276662 -0.472035
8 -0.013960 -0.362543
[9 rows x 2 columns]
In [36]: pd.reset_option("display.max_rows")
In [37]: pd.reset_option("display.min_rows")

display.expand_frame_repr allows for the representation of a DataFrame to stretch across pages, wrapped over the all the columns.

In [38]: df = pd.DataFrame(np.random.randn(5, 10))
In [39]: pd.set_option("expand_frame_repr", True)
In [40]: df
Out[40]:
   0   1   2   3   4   5   6   7   8
0 -0.006154 -0.923061 0.895717 0.805244 -1.206412 2.565646 1.431256 1.340309 -1.
1 -170299 -0.226169
2  0.410835  0.813850 0.132003 -0.827317 -0.076467 -1.187678 1.130127 -1.436737 -1.
3 -413681  1.607920
4  1.024180  0.569605 0.875906 -2.211372 0.974466 -2.006747 -0.410001 -0.078638 0.
5 -545952 -1.219217
(continues on next page)
In [41]: pd.set_option("expand_frame_repr", False)

In [42]: df
Out[42]:
     0    1    2    3   4    5    6    7  
0  -0.006154 -0.923061 0.895717 0.805244 -1.206412 2.565646 1.431256 1.340309
1    170299 -0.226169
2    0.410835 0.813850 0.132003 -0.827317 -0.076467 -1.187678 1.130127 -1.436737
3    0.1024180 0.569605 0.875906 -2.211372 0.974466 -2.006747 -0.410001 -0.078638
4    0.1024180 0.569605 0.875906 -2.211372 0.974466 -2.006747 -0.410001 -0.078638

In [43]: pd.reset_option("expand_frame_repr")

display.large_repr displays a DataFrame that exceed max_columns or max_rows as a truncated frame or summary.

In [44]: df = pd.DataFrame(np.random.randn(10, 10))

In [45]: pd.set_option("display.max_rows", 5)

In [46]: pd.set_option("large_repr", "truncate")

In [47]: df
Out[47]:
     0    1    2    3   4    5    6    7  
0  -0.954208 1.462696 -1.743161 -0.826591 -0.345352 1.314232 0.690579 0.995761
1    396780 0.014871
2    3.357427 -0.317441 -1.236269 0.896171 -0.487602 -0.082240 -2.182937 0.380396
3    0.432390
... ... ... ... ... ... ... ... ...
8    0.303421 -0.858447 0.306996 -0.028665 0.384316 1.574159 1.588931 0.476720
9    0.473424 -0.242861
[10 rows x 10 columns]

In [48]: pd.set_option("large_repr", "info")

(continues on next page)
In [49]: df
Out[49]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
   # Column   Non-Null Count   Dtype
   --- ------ -------------- -----  
   0    0     10 non-null   float64
   1    1     10 non-null   float64
   2    2     10 non-null   float64
   3    3     10 non-null   float64
   4    4     10 non-null   float64
   5    5     10 non-null   float64
   6    6     10 non-null   float64
   7    7     10 non-null   float64
   8    8     10 non-null   float64
   9    9     10 non-null   float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [50]: pd.reset_option("large_repr")

In [51]: pd.reset_option("display.max_rows")
display.max_colwidth sets the maximum width of columns. Cells of this length or longer will be truncated with an ellipsis.

In [52]: df = pd.DataFrame(
   ....:     np.array(
   ....:         ["foo", "bar", "bim", "uncomfortably long string"],
   ....:         ["horse", "cow", "banana", "apple"],
   ....:         )
   ....: )

In [53]: pd.set_option("max_colwidth", 40)

In [54]: df
Out[54]:
     0    1    2                                             3
0  foo  bar  bim  uncomfortably long string
1  horse  cow  banana  apple

In [55]: pd.set_option("max_colwidth", 6)

In [56]: df
Out[56]:
     0    1    2    3
0  foo  bar  bim  un...
1  horse  cow  ba...  apple
display.max_info_columns sets a threshold for the number of columns displayed when calling info().

```
In [58]: df = pd.DataFrame(np.random.randn(10, 10))
In [59]: pd.set_option("max_info_columns", 11)
In [60]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
 #   Column  Non-Null Count   Dtype
------  -------------- ------
 0   0   10 non-null float64
 1   1   10 non-null float64
 2   2   10 non-null float64
 3   3   10 non-null float64
 4   4   10 non-null float64
 5   5   10 non-null float64
 6   6   10 non-null float64
 7   7   10 non-null float64
 8   8   10 non-null float64
 9   9   10 non-null float64
dtypes: float64(10)
memory usage: 928.0 bytes
```

```
In [61]: pd.set_option("max_info_columns", 5)
In [62]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Columns: 10 entries, 0 to 9
dtypes: float64(10)
memory usage: 928.0 bytes
```

```
In [63]: pd.reset_option("max_info_columns")
```

display.max_info_rows: info() will usually show null-counts for each column. For a large DataFrame, this can be quite slow. max_info_rows and max_info_cols limit this null check to the specified rows and columns respectively. The info() keyword argument null_counts=True will override this.

```
In [64]: df = pd.DataFrame(np.random.choice([0, 1, np.nan], size=(10, 10)))
In [65]: df
```

```
Out[65]:
      0   1   2   3   4   5   6   7   8
0  0.0  NaN  NaN  NaN  0.0  NaN  NaN  1.0
1  1.0  NaN  1.0  1.0  1.0  NaN  0.0  0.0  NaN
2  0.0  NaN  1.0  0.0  0.0  NaN  NaN  NaN  0.0
3  NaN  NaN  NaN  0.0  1.0  1.0  NaN  1.0  NaN
```

(continues on next page)
In [66]: pd.set_option("max_info_rows", 11)

In [67]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
   #   Column    Non-Null Count   Dtype
---  ------      --------------   -----  
    0   0         8 non-null     float64
    1   1         3 non-null     float64
    2   2         7 non-null     float64
    3   3         6 non-null     float64
    4   4         7 non-null     float64
    5   5         6 non-null     float64
    6   6         2 non-null     float64
    7   7         6 non-null     float64
    8   8         6 non-null     float64
    9   9         6 non-null     float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [68]: pd.set_option("max_info_rows", 5)

In [69]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
   #   Column  Dtype
---  ------     -----  
    0   0 float64
    1   1 float64
    2   2 float64
    3   3 float64
    4   4 float64
    5   5 float64
    6   6 float64
    7   7 float64
    8   8 float64
    9   9 float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [70]: pd.reset_option("max_info_rows")

display.precision sets the output display precision in terms of decimal places.

2.22. Options and settings
In [71]: df = pd.DataFrame(np.random.randn(5, 5))

In [72]: pd.set_option("display.precision", 7)

In [73]: df
Out[73]:
   0     1     2     3     4
0 -1.1506 -0.7983 -0.5577  0.3814  1.3371
1 -1.5311  1.3315 -0.5713 -0.0267 -1.0857
2 -1.1147 -0.0582 -0.4868  1.6851  0.1126
3 -1.4953  0.8984 -1.4822 -1.5961  0.1597
4  0.2621  0.0362  0.1847 -0.2551 -0.2710

In [74]: pd.set_option("display.precision", 4)

In [75]: df
Out[75]:
   0     1     2     3     4
0 -1.1506 -0.7983 -0.5577  0.3814  1.3371
1 -1.5311  1.3315 -0.5713 -0.0267 -1.0857
2 -1.1147 -0.0582 -0.4868  1.6851  0.1126
3 -1.4953  0.8984 -1.4822 -1.5961  0.1597
4  0.2621  0.0362  0.1847 -0.2551 -0.2710

In [76]: df = pd.DataFrame(np.random.randn(6, 6))

In [77]: pd.set_option("chop_threshold", 0)

In [78]: df
Out[78]:
   0  1     2     3     4     5
0  1.2884  0.2946 -1.1658  0.8470 -0.6856  0.6091
1 -0.3040  0.6256 -0.0593  0.2497  1.1039 -1.0875
2  1.9980 -0.2445  0.1362  0.8863 -1.3507 -0.8863
3 -1.0133  1.9209 -0.3882 -2.3144  0.6655  0.4026
4  0.3996 -1.7660  0.8504  0.3881  0.9923  0.7441
5 -0.7398 -1.0549 -0.1796  0.6396  1.5850  1.9067

In [79]: pd.set_option("chop_threshold", 0.5)

In [80]: df
Out[80]:
   0  1     2     3     4     5
0  1.2884  0.0000 -1.1658  0.8470 -0.6856  0.6091
1  0.0000  0.6256  0.0000  0.0000  1.1039 -1.0875
2  1.9980  0.0000  0.0000  0.8863 -1.3507 -0.8863
3 -1.0133  1.9209  0.0000 -2.3144  0.6655  0.0000
4  0.0000 -1.7660  0.8504  0.0000  0.9923  0.7441
5 -0.7398 -1.0549  0.0000  0.6396  1.5850  1.9067

(continues on next page)
display.colheader_justify controls the justification of the headers. The options are 'right', and 'left'.

In [83]: df = pd.DataFrame(
        np.array([np.random.randn(6), np.random.randint(1, 9, 6) * 0.1, np.zeros(6)]).T,
        columns=["A", "B", "C"],
        dtype="float",
    )

In [84]: df
Out[84]:
         A    B    C
0  0.1040  0.1  0.0
1  0.1741  0.5  0.0
2 -0.4395  0.4  0.0
3 -0.7413  0.8  0.0
4 -0.0797  0.4  0.0
5 -0.9229  0.3  0.0

In [85]: pd.set_option("colheader_justify", "left")

In [86]: df
Out[86]:
         A    B    C
0  0.1040  0.1  0.0
1  0.1741  0.5  0.0
2 -0.4395  0.4  0.0
3 -0.7413  0.8  0.0
4 -0.0797  0.4  0.0
5 -0.9229  0.3  0.0

2.22.6 Number formatting

pandas also allows you to set how numbers are displayed in the console. This option is not set through the set_options API.

Use the set_eng_float_format function to alter the floating-point formatting of pandas objects to produce a particular format.

In [88]: import numpy as np

In [89]: pd.set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [90]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

(continues on next page)
In [91]: s / 1.0e3
Out[91]:
a 303.638u
b -721.084u
c -622.696u
d 648.250u
e -1.945m
dtype: float64

In [92]: s / 1.0e6
Out[92]:
a 303.638n
b -721.084n
c -622.696n
d 648.250n
e -1.945u
dtype: float64

Use `round()` to specifically control rounding of an individual `DataFrame`

### 2.22.7 Unicode formatting

**Warning:** Enabling this option will affect the performance for printing of DataFrame and Series (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters whose width corresponds to two Latin characters. If a DataFrame or Series contains these characters, the default output mode may not align them properly.

```python
In [93]: df = pd.DataFrame("": ["UK", ""], ": ["Alice", ""])

In [94]: df
Out[94]:
0     UK     Alice
1
```

Enabling `display.unicode.east_asian_width` allows pandas to check each character’s “East Asian Width” property. These characters can be aligned properly by setting this option to `True`. However, this will result in longer render times than the standard `len` function.

```python
In [95]: pd.set_option("display.unicode.east_asian_width", True)

In [96]: df
Out[96]:
0     UK     Alice
1
```

In addition, Unicode characters whose width is “ambiguous” can either be 1 or 2 characters wide depending on the...
terminal setting or encoding. The option display.unicode.ambiguous_as_wide can be used to handle the ambiguity.

By default, an “ambiguous” character’s width, such as “¡” (inverted exclamation) in the example below, is taken to be 1.

```python
In [97]: df = pd.DataFrame({"a": ["xxx", "¡¡"], "b": ["yyy", "¡¡"]})

In [98]: df
Out[98]:
   a    b
0  xxx  yyy
1  ¡¡   ¡¡
```

Enabling display.unicode.ambiguous_as_wide makes pandas interpret these characters’ widths to be 2. (Note that this option will only be effective when display.unicode.east_asian_width is enabled.)

However, setting this option incorrectly for your terminal will cause these characters to be aligned incorrectly:

```python
In [99]: pd.set_option("display.unicode.ambiguous_as_wide", True)

In [100]: df
Out[100]:
   a    b
0  xxx  yyy
1  ¡¡   ¡¡
```

### 2.22.8 Table schema display

DataFrame and Series will publish a Table Schema representation by default. This can be enabled globally with the display.html.table_schema option:

```python
In [101]: pd.set_option("display.html.table_schema", True)
```

Only 'display.max_rows' are serialized and published.

### 2.23 Enhancing performance

In this part of the tutorial, we will investigate how to speed up certain functions operating on pandas DataFrames using three different techniques: Cython, Numba and pandas.eval(). We will see a speed improvement of ~200 when we use Cython and Numba on a test function operating row-wise on the DataFrame. Using pandas.eval() we will speed up a sum by an order of ~2.

**Note:** In addition to following the steps in this tutorial, users interested in enhancing performance are highly encouraged to install the recommended dependencies for pandas. These dependencies are often not installed by default, but will offer speed improvements if present.
2.23.1 Cython (writing C extensions for pandas)

For many use cases writing pandas in pure Python and NumPy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in Python, for example by trying to remove for-loops and making use of NumPy vectorization. It’s always worth optimising in Python first.

This tutorial walks through a “typical” process of cythonizing a slow computation. We use an example from the Cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure Python solution.

Pure Python

We have a DataFrame to which we want to apply a function row-wise.

```
In [1]: df = pd.DataFrame(
    ...:     {
    ...:         "a": np.random.randn(1000),
    ...:         "b": np.random.randn(1000),
    ...:         "N": np.random.randint(100, 1000, (1000)),
    ...:         "x": "x",
    ...:     })

In [2]: df
Out[2]:
    a       b       N   x
  0  0.469112 -0.218470 585  x
  1 -0.282863 -0.061645 841  x
  2 -1.509059 -0.723780 251  x
  3 -1.135632  0.551225 972  x
  4  1.212112 -0.497767 181  x
  ... ... ... ... ...
995 -1.512743  0.874737 374  x
996  0.933753  1.120790 246  x
997 -0.308013  0.198768 157  x
998 -0.079915  1.757555 977  x
999 -1.010589 -1.115680 770  x
[1000 rows x 4 columns]
```

Here's the function in pure Python:

```
In [3]: def f(x):
    ...:     return x * (x - 1)

In [4]: def integrate_f(a, b, N):
    ...:     s = 0
    ...:     dx = (b - a) / N
    ...:     for i in range(N):
    ...:         s += f(a + i * dx)
```

(continues on next page)
We achieve our result by using `apply` (row-wise):

```python
In [7]: %timeit df.apply(lambda x: integrate_f(x["a"], x["b"], x["N"]), axis=1)
10 loops, best of 3: 174 ms per loop
```

But clearly this isn’t fast enough for us. Let’s take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the `prun` ipython magic function:

```text
638348 function calls (638330 primitive calls) in 0.146 seconds
Ordered by: internal time
List reduced from 228 to 4 due to restriction <4>
```

By far the majority of time is spend inside either `integrate_f` or `f`, hence we’ll concentrate our efforts cythonizing these two functions.

### Plain Cython

First we’re going to need to import the Cython magic function to IPython:

```text
In [6]: %load_ext Cython
```

Now, let’s simply copy our functions over to Cython as is (the suffix is here to distinguish between function versions):

```python
In [7]: %cython
def f_plain(x):
   ...
   return x * (x - 1)
def integrate_f_plain(a, b, N):
   ...
   s = 0
   ...
   dx = (b - a) / N
   ...
   for i in range(N):
      ...
      s += f_plain(a + i * dx)
   ...
   return s * dx
```

**Note:** If you’re having trouble pasting the above into your ipython, you may need to be using bleeding edge IPython for paste to play well with cell magics.
Adding type

We get another huge improvement simply by providing type information:

```python
In [8]: %cython
cdef double f_typed(double x) except? -2:
    ...
    return x * (x - 1)
cdef double integrate_f_typed(double a, double b, int N):
    ...
cdef int i
    ...
cdef double s, dx
    ...
s = 0
    ...
dx = (b - a) / N
    ...
    for i in range(N):
        ...
s += f_typed(a + i * dx)
    ...
    return s * dx
```

Numpy

Now, we're talking! It's now over ten times faster than the original Python implementation, and we haven't really modified the code. Let's have another look at what's eating up time:

```python
In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x["a"], x["b"], x["N"]), axis=1)
85918 function calls (85900 primitive calls) in 0.027 seconds
Ordered by: internal time
List reduced from 221 to 4 due to restriction <4>
ncalls tottime percall cumtime percall filename:lineno(function)
3000 0.004 0.000 0.017 0.000 series.py:943(__getitem__) 3000 0.002 0.000 0.010 0.000 series.py:1052(_get_value) 16189 0.002 0.000 0.002 0.000 {built-in method builtins.isinstance} 3000 0.001 0.000 0.005 0.000 base.py:3577(get_loc)
```

Using ndarray

It's calling series... a lot! It’s creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in Python, so maybe we could minimize these by cythonizing the apply part.

Note: We are now passing ndarrays into the Cython function, fortunately Cython plays very nicely with NumPy.
The implementation is simple, it creates an array of zeros and loops over the rows, applying our `integrate_f_typed`, and putting this in the zeros array.

**Warning:** You can **not pass** a Series directly as a ndarray typed parameter to a Cython function. Instead pass the actual ndarray using the `Series.to_numpy()`. The reason is that the Cython definition is specific to an ndarray and not the passed Series.

So, do not do this:

```
apply_integrate_f(df["a"], df["b"], df["N"])
```

But rather, use `Series.to_numpy()` to get the underlying ndarray:

```
apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].to_numpy())
```

**Note:** Loops like this would be extremely slow in Python, but in Cython looping over NumPy arrays is fast.

```
In [4]: %timeit apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].to_numpy())
1000 loops, best of 3: 1.25 ms per loop
```

We’ve gotten another big improvement. Let’s check again where the time is spent:

```
In [11]: %prun -l 4 apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].to_numpy())
200 function calls in 0.001 seconds
```
Ordered by: internal time
List reduced from 53 to 4 due to restriction <4>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>{built-in method _cython_magic_937fe8304e087ddfd94ad3b8c88783f6.apply_integrate_f}</td>
</tr>
<tr>
<td>3</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>frame.py:3463(<strong>getitem</strong>)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.001</td>
<td>{built-in method builtins.exec}</td>
</tr>
<tr>
<td>3</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>managers.py:1016(iget)</td>
</tr>
</tbody>
</table>

As one might expect, the majority of the time is now spent in apply_integrate_f, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

**More advanced techniques**

There is still hope for improvement. Here’s an example of using some more advanced Cython techniques:

```cython
.. cimport cython
.. cimport numpy as np
.. import numpy as np

cdef double f_typed(double x) except? -2:
    return x * (x - 1)
cdef double integrate_f_typed(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_typed(a + i * dx)
    return s * dx

@cython.boundscheck(False)
@cython.wraparound(False)
cdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a,
                        np.ndarray[double] col_b,
                        np.ndarray[int] col_N):
    cdef int i, n = len(col_N)
    assert len(col_a) == len(col_b) == n
    cdef np.ndarray[double] res = np.empty(n)
    for i in range(n):
        res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
    return res
```

```python
In [4]: %timeit apply_integrate_f_wrap(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].
to_numpy())
10000 loops, best of 3: 987 us per loop
```

Even faster, with the caveat that a bug in our Cython code (an off-by-one error, for example) might cause a segfault because memory access isn’t checked. For more about boundscheck and wraparound, see the Cython docs on compiler directives.
2.23.2 Numba (JIT compilation)

An alternative to statically compiling Cython code is to use a dynamic just-in-time (JIT) compiler with Numba.

Numba allows you to write a pure Python function which can be JIT compiled to native machine instructions, similar in performance to C, C++ and Fortran, by decorating your function with `@jit`.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pyc code). Numba supports compilation of Python to run on either CPU or GPU hardware and is designed to integrate with the Python scientific software stack.

**Note:** The `@jit` compilation will add overhead to the runtime of the function, so performance benefits may not be realized especially when using small data sets. Consider caching your function to avoid compilation overhead each time your function is run.

Numba can be used in 2 ways with pandas:

1. Specify the `engine="numba"` keyword in select pandas methods
2. Define your own Python function decorated with `@jit` and pass the underlying NumPy array of `Series` or `Dataframe` (using `to_numpy()`) into the function

**pandas Numba Engine**

If Numba is installed, one can specify `engine="numba"` in select pandas methods to execute the method using Numba. Methods that support `engine="numba"` will also have an `engine_kwargs` keyword that accepts a dictionary that allows one to specify "nogil", "nopython" and "parallel" keys with boolean values to pass into the `@jit` decorator. If `engine_kwargs` is not specified, it defaults to `{"nogil": False, "nopython": True, "parallel": False}` unless otherwise specified.

In terms of performance, the first time a function is run using the Numba engine will be slow as Numba will have some function compilation overhead. However, the JIT compiled functions are cached, and subsequent calls will be fast. In general, the Numba engine is performant with a larger amount of data points (e.g. 1+ million).

```
In [1]: data = pd.Series(range(1_000_000))  # noqa: E225
In [2]: roll = data.rolling(10)
In [3]: def f(x):
   ...:     return np.sum(x) + 5
   # Run the first time, compilation time will affect performance
In [4]: %timeit -r 1 -n 1 roll.apply(f, engine='numba', raw=True)
1.23 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
   # Function is cached and performance will improve
In [5]: %timeit roll.apply(f, engine='numba', raw=True)
188 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [6]: %timeit roll.apply(f, engine='cython', raw=True)
3.92 s ± 59 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```
**Custom Function Examples**

A custom Python function decorated with `@jit` can be used with pandas objects by passing their NumPy array representations with `to_numpy()`.

```python
import numba

@numba.jit
def f_plain(x):
    return x ** (x - 1)

@numba.jit
def integrate_f_numba(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_plain(a + i * dx)
    return s * dx

@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
    n = len(col_N)
    result = np.empty(n, dtype="float64")
    assert len(col_a) == len(col_b) == n
    for i in range(n):
        result[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
    return result

def compute_numba(df):
    result = apply_integrate_f_numba(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].to_numpy())
    return pd.Series(result, index=df.index, name="result")
```

In [4]: `%timeit` compute_numba(df)
1000 loops, best of 3: 798 us per loop

In this example, using Numba was faster than Cython.

Numba can also be used to write vectorized functions that do not require the user to explicitly loop over the observations of a vector; a vectorized function will be applied to each row automatically. Consider the following example of doubling each observation:

```python
import numba

def double_every_value_nonumba(x):
    return x * 2
```

(continues on next page)
@numba.vectorize
def double_every_value_withnumba(x): # noqa E501
    return x * 2

# Custom function without numba
In [5]: %timeit df["coll_doubled"] = df["a"].apply(double_every_value_nonumba) # noqa.._E501
1000 loops, best of 3: 797 us per loop

# Standard implementation (faster than a custom function)
In [6]: %timeit df["coll_doubled"] = df["a"] * 2
1000 loops, best of 3: 233 us per loop

# Custom function with numba
In [7]: %timeit df["coll_doubled"] = double_every_value_withnumba(df["a"].to_numpy())
1000 loops, best of 3: 145 us per loop

Caveats

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. If you try to @jit a function that contains unsupported Python or NumPy code, compilation will revert object mode which will mostly likely not speed up your function. If you would prefer that Numba throw an error if it cannot compile a function in a way that speeds up your code, pass Numba the argument nopython=True (e.g. @jit(nopython=True)). For more on troubleshooting Numba modes, see the Numba troubleshooting page.

Using parallel=True (e.g. @jit(parallel=True)) may result in a SIGABRT if the threading layer leads to unsafe behavior. You can first specify a safe threading layer before running a JIT function with parallel=True.

Generally if the you encounter a segfault (SIGSEGV) while using Numba, please report the issue to the Numba issue tracker.

2.23.3 Expression evaluation via eval()

The top-level function pandas.eval() implements expression evaluation of Series and DataFrame objects.

Note: To benefit from using eval() you need to install numexpr. See the recommended dependencies section for more details.

The point of using eval() for expression evaluation rather than plain Python is two-fold: 1) large DataFrame objects are evaluated more efficiently and 2) large arithmetic and boolean expressions are evaluated all at once by the underlying engine (by default numexpr is used for evaluation).

Note: You should not use eval() for simple expressions or for expressions involving small DataFrames. In fact, eval() is many orders of magnitude slower for smaller expressions/objects than plain ol’ Python. A good rule of thumb is to only use eval() when you have a DataFrame with more than 10,000 rows.

eval() supports all arithmetic expressions supported by the engine in addition to some extensions available only in pandas.
Note: The larger the frame and the larger the expression the more speedup you will see from using `eval()`.

Supported syntax

These operations are supported by `pandas.eval()`:

- Arithmetic operations except for the left shift (<<) and right shift (>>) operators, e.g., `df + 2 * pi / s ** 4 % 42 - the_golden_ratio`
- Comparison operations, including chained comparisons, e.g., `2 < df < df2`
- Boolean operations, e.g., `df < df2 and df3 < df4 or not df_bool`
- list and tuple literals, e.g., `[1, 2]` or `(1, 2)`
- Attribute access, e.g., `df.a`
- Subscript expressions, e.g., `df[0]`
- Simple variable evaluation, e.g., `pd.eval("df")` (this is not very useful)
- Math functions: `sin`, `cos`, `exp`, `log`, `expm1`, `log1p`, `sqrt`, `sinh`, `cosh`, `tanh`, `arcsin`, `arccos`, `arctan`, `arccosh`, `arcsinh`, `arctanh`, `abs`, `arctan2` and `log10`.

This Python syntax is not allowed:

- Expressions
  - Function calls other than math functions.
  - `is/is not` operations
  - `if` expressions
  - `lambda` expressions
  - list/set/dict comprehensions
  - Literal dict and set expressions
  - `yield` expressions
  - Generator expressions
  - Boolean expressions consisting of only scalar values
- Statements
  - Neither `simple` nor `compound` statements are allowed. This includes things like `for`, `while`, and `if`.

`eval()` examples

`pandas.eval()` works well with expressions containing large arrays.

First let’s create a few decent-sized arrays to play with:

```
In [13]: nrows, ncols = 20000, 100
In [14]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols)) for _ in range(4)]
```

Now let’s compare adding them together using plain ol’ Python versus `eval()`:
In [15]: %timeit df1 + df2 + df3 + df4
10.2 ms +- 236 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [16]: %timeit pd.eval("df1 + df2 + df3 + df4")
4.38 ms +- 249 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

Now let's do the same thing but with comparisons:

In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
5.01 ms +- 29.9 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [18]: %timeit pd.eval("(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)")
4.6 ms +- 193 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

eval() also works with unaligned pandas objects:

In [19]: s = pd.Series(np.random.randn(50))

In [20]: %timeit df1 + df2 + df3 + df4 + s
17.5 ms +- 618 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [21]: %timeit pd.eval("df1 + df2 + df3 + df4 + s")
5.15 ms +- 132 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

Note: Operations such as

```python
1 and 2  # would parse to 1 & 2, but should evaluate to 2
3 or 4  # would parse to 3 | 4, but should evaluate to 3
~1  # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type bool or np.bool_. Again, you should perform these kinds of operations in plain Python.

The DataFrame.eval method

In addition to the top level pandas.eval() function you can also evaluate an expression in the “context” of a DataFrame.

In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=["a", "b"])

In [23]: df.eval("a + b")
Out[23]:
0   -0.246747
1    0.867786
2   -1.626063
3   -1.134978
4   -1.027798
dtype: float64
Any expression that is a valid `pandas.eval()` expression is also a valid `DataFrame.eval()` expression, with the added benefit that you don’t have to prefix the name of the `DataFrame` to the column(s) you’re interested in evaluating.

In addition, you can perform assignment of columns within an expression. This allows for formulaic evaluation. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

The `inplace` keyword determines whether this assignment will performed on the original DataFrame or return a copy with the new column.

```python
In [24]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
In [25]: df.eval("c = a + b", inplace=True)
In [26]: df.eval("d = a + b + c", inplace=True)
In [27]: df.eval("a = 1", inplace=True)
In [28]: df
Out[28]:
   a  b  c  d
0  0  5  5 10
1  1  6  7 14
2  2  7  9 18
3  3  8 11 22
4  4  9 13 26

In [29]: df
Out[29]:
   a  b  c  d
0  0  5  5 10
1  1  6  7 14
2  2  7  9 18
3  3  8 11 22
4  4  9 13 26

In [30]: df.eval("e = a - c", inplace=False)
Out[30]:
   a  b  c  d  e
0  0  5  5 10  -4
1  1  6  7 14  -6
2  2  7  9 18  -8
3  3  8 11 22 -10
4  4  9 13 26 -12

In [31]: df
Out[31]:
   a  b  c  d
0  0  5  5 10
1  1  6  7 14
2  2  7  9 18
3  3  8 11 22
4  4  9 13 26
```

When `inplace` is set to `False`, the default, a copy of the DataFrame with the new or modified columns is returned and the original frame is unchanged.
As a convenience, multiple assignments can be performed by using a multi-line string.

```python
In [32]: df.eval(
    .....: c = a + b
    .....: d = a + b + c
    .....: a = 1
    .....: inplace=False,
    .....: )
```

```
Out[32]:
   a  b  c  d
0  1  5  6  12
1  1  6  7  14
2  1  7  8  16
3  1  8  9  18
4  1  9  10 20
```

The equivalent in standard Python would be

```python
In [33]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [34]: df["c"] = df["a"] + df["b"]

In [35]: df["d"] = df["a"] + df["b"] + df["c"]

In [36]: df["a"] = 1

In [37]: df
```

```
Out[37]:
   a  b  c  d
0  1  5  5  10
1  1  6  7  14
2  1  7  9  18
3  1  8 11  22
4  1  9 13  26
```

The query method has an `inplace` keyword which determines whether the query modifies the original frame.

```python
In [38]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [39]: df.query("a > 2")
```

```
Out[39]:
   a  b
3  3  8
4  4  9
```

```python
In [40]: df.query("a > 2", inplace=True)

In [41]: df
```

```
Out[41]:
   a  b
3  3  8
4  4  9
```
Local variables

You must *explicitly reference* any local variable that you want to use in an expression by placing the `@` character in front of the name. For example,

```python
In [42]: df = pd.DataFrame(np.random.randn(5, 2), columns=list("ab"))

In [43]: newcol = np.random.randn(len(df))

In [44]: df.eval("b + @newcol")
Out[44]:
   0  -0.173926
   1   2.493083
   2   0.881831
   3  -0.691045
   4   1.334703
   dtype: float64

In [45]: df.query("b < @newcol")
Out[45]:
   a   b
  0  0.863987 -0.115998
  2 -2.621419 -1.297879
```

If you don’t prefix the local variable with `@`, pandas will raise an exception telling you the variable is undefined.

When using `DataFrame.eval()` and `DataFrame.query()`, this allows you to have a local variable and a `DataFrame` column with the same name in an expression.

```python
In [46]: a = np.random.randn()

In [47]: df.query("@a < a")
Out[47]:
   a   b
  0  0.863987 -0.115998
  2 -2.621419 -1.297879

In [48]: df.loc[a < df["a"]]
   # same as the previous expression
Out[48]:
   a   b
  0  0.863987 -0.115998
```

With `pandas.eval()` you cannot use the `@` prefix *at all*, because it isn’t defined in that context. pandas will let you know this if you try to use `@` in a top-level call to `pandas.eval()`. For example,

```python
In [49]: a, b = 1, 2

In [50]: pd.eval("@a + b")
Traceback (most recent call last):
    File /opt/conda/envs/pandas/lib/python3.8/site-packages/IPython/core/interactiveshell.py:3369 in run_code
      exec(code_obj, self.user_global_ns, self.user_ns)
In [50] in <cell line: 1>
```
In this case, you should simply refer to the variables like you would in standard Python.

```
in [51]: pd.eval("a + b")
Out[51]: 3
```

### pandas.eval() parsers

There are two different parsers and two different engines you can use as the backend.

The default 'pandas' parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the & and | operators is made equal to the precedence of the corresponding boolean operations and or.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the 'python' parser to enforce strict Python semantics.

```
in [52]: expr = "(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)"
In [53]: x = pd.eval(expr, parser="python")
In [54]: expr_no_parens = "df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0"
In [55]: y = pd.eval(expr_no_parens, parser="pandas")
In [56]: np.all(x == y)
Out[56]: True
```

The same expression can be “anded” together with the word and as well:

```
in [57]: expr = "(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)"
In [58]: x = pd.eval(expr, parser="python")
In [59]: expr_with_ands = "df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0"
In [60]: y = pd.eval(expr_with_ands, parser="pandas")
In [61]: np.all(x == y)
Out[61]: True
```

The and and or operators here have the same precedence that they would in vanilla Python.
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.eval() backends**

There’s also the option to make `eval()` operate identical to plain ol’ Python.

**Note:** Using the ‘python’ engine is generally *not* useful, except for testing other evaluation engines against it. You will achieve no performance benefits using `eval()` with `engine='python'` and in fact may incur a performance hit.

You can see this by using `pandas.eval()` with the 'python' engine. It is a bit slower (not by much) than evaluating the same expression in Python:

```python
In [62]: %timeit df1 + df2 + df3 + df4
10.4 ms +- 134 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```python
In [63]: %timeit pd.eval("df1 + df2 + df3 + df4", engine="python")
11.3 ms +- 274 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

**pandas.eval() performance**

`eval()` is intended to speed up certain kinds of operations. In particular, those operations involving complex expressions with large `DataFrame/Series` objects should see a significant performance benefit. Here is a plot showing the running time of `pandas.eval()` as function of the size of the frame involved in the computation. The two lines are two different engines.

![DataFrame.eval()](image)

**Note:** Operations with smallish objects (around 15k-20k rows) are faster using plain Python:
This plot was created using a DataFrame with 3 columns each containing floating point values generated using \texttt{numpy.random.randn()}. 

\textbf{Technical minutia regarding expression evaluation}

Expressions that would result in an object dtype or involve datetime operations (because of NaT) must be evaluated in Python space. The main reason for this behavior is to maintain backwards compatibility with versions of NumPy < 1.7. In those versions of NumPy a call to \texttt{ndarray.astype(str)} will truncate any strings that are more than 60 characters in length. Second, we can’t pass object arrays to \texttt{numexpr} thus string comparisons must be evaluated in Python space.

The upshot is that this only applies to object-dtype expressions. So, if you have an expression—for example

\begin{verbatim}
In [64]: df = pd.DataFrame(
        .....:     {"strings": np.repeat(list("cba"), 3), "nums": np.repeat(range(3), 3)}
        .....: )
        .....:
In [65]: df
Out[65]:
   strings  nums
   0      c   0
   1      c   0
   2      c   0
   3      b   1
   4      b   1
   5      b   1
   6      a   2
   7      a   2
   8      a   2
In [66]: df.query("strings == 'a' and nums == 1")
Out[66]:
\end{verbatim}

(continues on next page)
the numeric part of the comparison (nums == 1) will be evaluated by numexpr.

In general, `DataFrame.query()`/`pandas.eval()` will evaluate the subexpressions that can be evaluated by numexpr and those that must be evaluated in Python space transparently to the user. This is done by inferring the result type of an expression from its arguments and operators.

### 2.24 Scaling to large datasets

pandas provides data structures for in-memory analytics, which makes using pandas to analyze datasets that are larger than memory datasets somewhat tricky. Even datasets that are a sizable fraction of memory become unwieldy, as some pandas operations need to make intermediate copies.

This document provides a few recommendations for scaling your analysis to larger datasets. It’s a complement to *Enhancing performance*, which focuses on speeding up analysis for datasets that fit in memory.

But first, it’s worth considering *not using pandas.* pandas isn’t the right tool for all situations. If you’re working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let’s carry on.

```
In [1]: import pandas as pd
In [2]: import numpy as np
```

#### 2.24.1 Load less data

Suppose our raw dataset on disk has many columns:

```
     id_0 name_0  x_0  y_0  id_1 name_1  x_1  y_1
  ... name_8  x_8  y_8  id_9 name_9  x_9  y_9

timestamp...
2000-01-01 00:00:00 1015 Michael -0.399453 0.095427 994 Frank -0.176842 ...
  ...  ...  ...  ...  ...  ...  ...  ...
2000-01-01 00:01:00 969 Patricia 0.650773 -0.874275 1003 Laura 0.459153 ...
2000-01-01 00:02:00 1016 Victor -0.721465 -0.584710 1046 Michael 0.524994 ...
  ...  ...  ...  ...  ...  ...  ...  ...
2000-12-30 23:56:00 999 Tim 0.162578 0.512817 973 Kevin -0.403352 ...
  ...  ...  ...  ...  ...  ...  ...  ...
2000-12-30 23:57:00 970 Laura -0.435868 -0.600289 958 Oliver -0.966577 ...
  ...  ...  ...  ...  ...  ...  ...  ...
2000-12-30 23:58:00 1065 Edith 0.232211 -0.454540 971 Tim 0.158484 ...
  ...  ...  ...  ...  ...  ...  ...  ...
```

(continues on next page)
To load the columns we want, we have two options. Option 1 loads in all the data and then filters to what we need.

```
In [3]: columns = ["id_0", "name_0", "x_0", "y_0"]

In [4]: pd.read_parquet("timeseries_wide.parquet")[columns]
Out[4]:

<table>
<thead>
<tr>
<th>timestamp</th>
<th>id_0</th>
<th>name_0</th>
<th>x_0</th>
<th>y_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01 00:00:00</td>
<td>1015</td>
<td>Michael</td>
<td>-0.399453</td>
<td>0.095427</td>
</tr>
<tr>
<td>2000-01-01 00:01:00</td>
<td>969</td>
<td>Patricia</td>
<td>0.650773</td>
<td>-0.874275</td>
</tr>
<tr>
<td>2000-01-01 00:02:00</td>
<td>1016</td>
<td>Victor</td>
<td>-0.721465</td>
<td>-0.584710</td>
</tr>
<tr>
<td>2000-01-01 00:03:00</td>
<td>939</td>
<td>Alice</td>
<td>-0.746004</td>
<td>-0.908008</td>
</tr>
<tr>
<td>2000-01-01 00:04:00</td>
<td>1017</td>
<td>Dan</td>
<td>0.919451</td>
<td>-0.803504</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000-12-30 23:56:00</td>
<td>999</td>
<td>Tim</td>
<td>0.162578</td>
<td>0.512817</td>
</tr>
<tr>
<td>2000-12-30 23:57:00</td>
<td>970</td>
<td>Laura</td>
<td>-0.433586</td>
<td>-0.600289</td>
</tr>
<tr>
<td>2000-12-30 23:58:00</td>
<td>1065</td>
<td>Edith</td>
<td>0.232211</td>
<td>-0.454540</td>
</tr>
<tr>
<td>2000-12-30 23:59:00</td>
<td>1019</td>
<td>Ingrid</td>
<td>0.322208</td>
<td>-0.615974</td>
</tr>
<tr>
<td>2000-12-31 00:00:00</td>
<td>937</td>
<td>Ursula</td>
<td>-0.906523</td>
<td>0.943178</td>
</tr>
</tbody>
</table>

[525601 rows x 4 columns]
```

Option 2 only loads the columns we request.

```
In [5]: pd.read_parquet("timeseries_wide.parquet", columns=columns)
Out[5]:

<table>
<thead>
<tr>
<th>timestamp</th>
<th>id_0</th>
<th>name_0</th>
<th>x_0</th>
<th>y_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01 00:00:00</td>
<td>1015</td>
<td>Michael</td>
<td>-0.399453</td>
<td>0.095427</td>
</tr>
<tr>
<td>2000-01-01 00:01:00</td>
<td>969</td>
<td>Patricia</td>
<td>0.650773</td>
<td>-0.874275</td>
</tr>
<tr>
<td>2000-01-01 00:02:00</td>
<td>1016</td>
<td>Victor</td>
<td>-0.721465</td>
<td>-0.584710</td>
</tr>
<tr>
<td>2000-01-01 00:03:00</td>
<td>939</td>
<td>Alice</td>
<td>-0.746004</td>
<td>-0.908008</td>
</tr>
<tr>
<td>2000-01-01 00:04:00</td>
<td>1017</td>
<td>Dan</td>
<td>0.919451</td>
<td>-0.803504</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000-12-30 23:56:00</td>
<td>999</td>
<td>Tim</td>
<td>0.162578</td>
<td>0.512817</td>
</tr>
<tr>
<td>2000-12-30 23:57:00</td>
<td>970</td>
<td>Laura</td>
<td>-0.433586</td>
<td>-0.600289</td>
</tr>
<tr>
<td>2000-12-30 23:58:00</td>
<td>1065</td>
<td>Edith</td>
<td>0.232211</td>
<td>-0.454540</td>
</tr>
<tr>
<td>2000-12-30 23:59:00</td>
<td>1019</td>
<td>Ingrid</td>
<td>0.322208</td>
<td>-0.615974</td>
</tr>
<tr>
<td>2000-12-31 00:00:00</td>
<td>937</td>
<td>Ursula</td>
<td>-0.906523</td>
<td>0.943178</td>
</tr>
</tbody>
</table>

[525601 rows x 4 columns]
```

If we were to measure the memory usage of the two calls, we’d see that specifying columns uses about 1/10th the memory in this case.

With `pandas.read_csv()`, you can specify `usecols` to limit the columns read into memory. Not all file formats that can be read by pandas provide an option to read a subset of columns.

2.24. Scaling to large datasets
2.24.2 Use efficient datatypes

The default pandas data types are not the most memory efficient. This is especially true for text data columns with relatively few unique values (commonly referred to as “low-cardinality” data). By using more efficient data types, you can store larger datasets in memory.

```python
In [6]: ts = pd.read_parquet("timeseries.parquet")

In [7]: ts
Out[7]:
   id  name   x       y
   timestamp
2000-01-01 00:00:00 1029  Michael  0.278837  0.247932
2000-01-01 00:00:30 1010 Patricia  0.077144  0.490260
2000-01-01 00:01:00 1001  Victor   0.214525  0.258635
2000-01-01 00:01:30 1018   Alice -0.646866  0.822104
2000-01-01 00:02:00   991    Dan    0.902389  0.466665
                    ...   ...     ...       ...       ...
2000-12-30 23:58:00  992 Sarah    0.721155  0.944118
2000-12-30 23:58:30 1007  Ursula   0.409277  0.133227
2000-12-30 23:59:00 1009  Hannah  -0.452802  0.184318
2000-12-30 23:59:30  978   Kevin  -0.904728 -0.179146
2000-12-31 00:00:00   973  Ingrid -0.370763 -0.794667
[1051201 rows x 4 columns]
```

Now, let's inspect the data types and memory usage to see where we should focus our attention.

```python
In [8]: ts.dtypes
Out[8]:
   id     int64
  name   object
       x   float64
       y   float64
dtype: object

In [9]: ts.memory_usage(deep=True)  # memory usage in bytes
Out[9]:
   Index   8409608
          id   8409608
          name   65537768
          x   8409608
          y   8409608
dtype: int64
```

The `name` column is taking up much more memory than any other. It has just a few unique values, so it’s a good candidate for converting to a Categorical. With a Categorical, we store each unique name once and use space-efficient integers to know which specific name is used in each row.

```python
In [10]: ts2 = ts.copy()

In [11]: ts2["name"] = ts2["name"].astype("category")
```

(continues on next page)
We can go a bit further and downcast the numeric columns to their smallest types using `pandas.to_numeric()`.

```python
In [13]: ts2["id"] = pd.to_numeric(ts2["id"], downcast="unsigned")
```

```python
In [14]: ts2[["x", "y"]].apply(pd.to_numeric, downcast="float")
```

```python
In [15]: ts2.dtypes
```

```
Out[15]:
            id    name       x      y
dtype: object
```

```python
In [16]: ts2.memory_usage(deep=True)
```

```
Out[16]:
            Index  id  name  x  y
dtype: int64
```

```python
In [17]: reduction = ts2.memory_usage(deep=True).sum() / ts.memory_usage(deep=True).sum()
```

```python
In [18]: print(f"{reduction:.2f}")
```

```
0.20
```

In all, we’ve reduced the in-memory footprint of this dataset to 1/5 of its original size.

See `Categorical data` for more on `Categorical` and `dtypes` for an overview of all of pandas’ dtypes.

### 2.24.3 Use chunking

Some workloads can be achieved with chunking: splitting a large problem like “convert this directory of CSVs to parquet” into a bunch of small problems (“convert this individual CSV file into a Parquet file. Now repeat that for each file in this directory.”). As long as each chunk fits in memory, you can work with datasets that are much larger than memory.

**Note:** Chunking works well when the operation you’re performing requires zero or minimal coordination between chunks. For more complicated workflows, you’re better off using another library.
Suppose we have an even larger “logical dataset” on disk that’s a directory of parquet files. Each file in the directory represents a different year of the entire dataset.

```python
In [19]: %%time
   ....: files = pathlib.Path("data/timeseries/").glob("ts*.parquet")
   ....: counts = pd.Series(dtype=int)
   ....: for path in files:
   ....:     df = pd.read_parquet(path)
   ....:     counts = counts.add(df["name"].value_counts(), fill_value=0)
   ....: counts.astype(int)
   ....:
CPU times: user 481 ms, sys: 98 ms, total: 579 ms
Wall time: 542 ms
Out[19]:
Alice 229802
Bob 229211
Charlie 229303
Dan 230621
Edith 230349
... Victor 230502
Wendy 230038
Xavier 229553
Yvonne 228766
Zelda 229909
Length: 26, dtype: int64
```

Some readers, like `pandas.read_csv()`, offer parameters to control the `chunksize` when reading a single file.

Manually chunking is an OK option for workflows that don’t require too sophisticated of operations. Some operations, like `groupby`, are much harder to do chunkwise. In these cases, you may be better switching to a different library that implements these out-of-core algorithms for you.
2.24.4 Use other libraries

pandas is just one library offering a DataFrame API. Because of its popularity, pandas' API has become something of a standard that other libraries implement. The pandas documentation maintains a list of libraries implementing a DataFrame API in our ecosystem page.

For example, Dask, a parallel computing library, has `dask.dataframe`, a pandas-like API for working with larger than memory datasets in parallel. Dask can use multiple threads or processes on a single machine, or a cluster of machines to process data in parallel.

We'll import `dask.dataframe` and notice that the API feels similar to pandas. We can use Dask's `read_parquet` function, but provide a globstring of files to read in.

```
In [20]: import dask.dataframe as dd

In [21]: ddf = dd.read_parquet("data/timeseries/ts*.parquet", engine="pyarrow")

In [22]: ddf
Out[22]:
Dask DataFrame Structure:
    id   name    x      y
npartitions=12
       int64  object  float64  float64
...   ...     ...    ...    ...
...   ...     ...    ...    ...
...   ...     ...    ...    ...
...   ...     ...    ...    ...
Dask Name: read-parquet, 12 tasks
```

Inspecting the `ddf` object, we see a few things

- There are familiar attributes like `.columns` and `.dtypes`
- There are familiar methods like `.groupby`, `.sum`, etc.
- There are new attributes like `.npartitions` and `.divisions`

The partitions and divisions are how Dask parallelizes computation. A Dask DataFrame is made up of many pandas DataFrames. A single method call on a Dask DataFrame ends up making many pandas method calls, and Dask knows how to coordinate everything to get the result.

```
In [23]: ddf.columns
Out[23]: Index(['id', 'name', 'x', 'y'], dtype='object')

In [24]: ddf.dtypes
Out[24]:
    id      int64
name   object
    x      float64
    y      float64
dtype: object

In [25]: ddf.npartitions
Out[25]: 12
```

One major difference: the `dask.dataframe` API is lazy. If you look at the repr above, you'll notice that the values aren't actually printed out; just the column names and dtypes. That's because Dask hasn't actually read the data yet.

2.24. Scaling to large datasets
Rather than executing immediately, doing operations build up a **task graph**.

```
In [26]: ddf
Out[26]:  
Dask DataFrame Structure:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int64</td>
<td>object</td>
<td>float64</td>
<td>float64</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

npartitions=12
Dask Name: read-parquet, 12 tasks
```

```
In [27]: ddf["name"]
Out[27]:  
Dask Series Structure:

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

npartitions=12
object
... 
... 
... 
... 

Name: name, dtype: object
Dask Name: getitem, 24 tasks
```

```
In [28]: ddf["name"].value_counts()
Out[28]:  
Dask Series Structure:

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>int64</td>
</tr>
</tbody>
</table>
| ... 
| ... 
| ... 
| ... 

npartitions=1
Dask Name: value-counts-agg, 39 tasks
```

Each of these calls is instant because the result isn’t being computed yet. We’re just building up a list of computation to do when someone needs the result. Dask knows that the return type of a `pandas.Series.value_counts` is a pandas Series with a certain dtype and a certain name. So the Dask version returns a Dask Series with the same dtype and the same name.

To get the actual result you can call `.compute()`.

```
In [29]: %time ddf["name"].value_counts().compute()
CPU times: user 691 ms, sys: 233 ms, total: 924 ms
Wall time: 441 ms
Out[29]:  
Laura  230906
Ingrid 230838
Kevin  230698
Dan    230621
Frank  230595
...   
Ray    229603
Xavier 229553
```

(continues on next page)
At that point, you get back the same thing you’d get with pandas, in this case a concrete pandas Series with the count of each name.

Calling `.compute` causes the full task graph to be executed. This includes reading the data, selecting the columns, and doing the `value_counts`. The execution is done in parallel where possible, and Dask tries to keep the overall memory footprint small. You can work with datasets that are much larger than memory, as long as each partition (a regular pandas DataFrame) fits in memory.

By default, `dask.dataframe` operations use a threadpool to do operations in parallel. We can also connect to a cluster to distribute the work on many machines. In this case we’ll connect to a local “cluster” made up of several processes on this single machine.

```python
>>> from dask.distributed import Client, LocalCluster
>>> cluster = LocalCluster()
>>> client = Client(cluster)
>>> client
<Client: 'tcp://127.0.0.1:53349' processes=4 threads=8, memory=17.18 GB>
```

Once this `client` is created, all of Dask’s computation will take place on the cluster (which is just processes in this case).

Dask implements the most used parts of the pandas API. For example, we can do a familiar groupby aggregation.

```python
In [30]: %time ddf.groupby("name")[["x", "y"]].mean().compute().head()
CPU times: user 1.53 s, sys: 313 ms, total: 1.84 s
Wall time: 717 ms
```

```
Out[30]:
         x      y
name
Alice  0.000086 -0.001170
Bob   -0.000843 -0.000799
Charlie  0.000564 -0.000038
Dan    0.000584  0.000818
Edith  -0.000116 -0.000044
```

The grouping and aggregation is done out-of-core and in parallel.

When Dask knows the `divisions` of a dataset, certain optimizations are possible. When reading parquet datasets written by dask, the divisions will be known automatically. In this case, since we created the parquet files manually, we need to supply the divisions manually.

```python
In [31]: N = 12

In [32]: starts = [f"20{i:02d}-01-01" for i in range(N)]
In [33]: ends = [f"20{i:02d}-12-13" for i in range(N)]
In [34]: divisions = tuple(pd.to_datetime(starts)) + (pd.Timestamp(ends[-1]),)
```

(continues on next page)
In [35]: ddf.divisions = divisions

In [36]: ddf
Out[36]:
Dask DataFrame Structure:
   id   name   x   y
npartitions=12
2000-01-01  int64  object  float64  float64
2001-01-01  ...  ...  ...  ...  ...
...  ...  ...  ...  ...  ...
2011-01-01  ...  ...  ...  ...  ...
2011-12-13  ...  ...  ...  ...  ...
Dask Name: read-parquet, 12 tasks

Now we can do things like fast random access with .loc.

In [37]: ddf.loc["2002-01-01 12:01":"2002-01-01 12:05"].compute()
Out[37]:
   id   name    x      y
timestamp
2002-01-01 12:01:00  983  Laura  0.243985  -0.079392
2002-01-01 12:02:00 1001  Laura -0.523119  -0.226026
2002-01-01 12:03:00 1059  Oliver  0.612886   0.405680
2002-01-01 12:04:00  993   Kevin  0.451977   0.332947
2002-01-01 12:05:00 1014  Yvonne -0.948681   0.361748

Dask knows to just look in the 3rd partition for selecting values in 2002. It doesn’t need to look at any other data.

Many workflows involve a large amount of data and processing it in a way that reduces the size to something that fits in memory. In this case, we’ll resample to daily frequency and take the mean. Once we’ve taken the mean, we know the results will fit in memory, so we can safely call compute without running out of memory. At that point it’s just a regular pandas object.

In [38]: ddf[["x", "y"]].resample("1D").mean().cumsum().compute().plot()
These Dask examples have all been done using multiple processes on a single machine. Dask can be deployed on a cluster to scale up to even larger datasets.

You see more Dask examples at https://examples.dask.org.

### 2.25 Sparse data structures

```
In [1]: arr = np.random.randn(10)
In [2]: arr[2:-2] = np.nan
In [3]: ts = pd.Series(pd.arrays.SparseArray(arr))
In [4]: ts
Out[4]:
0    0.469112
1   -0.282863
```

*pandas* provides data structures for efficiently storing sparse data. These are not necessarily sparse in the typical “mostly 0”. Rather, you can view these objects as being “compressed” where any data matching a specific value (NaN / missing value, though any value can be chosen, including 0) is omitted. The compressed values are not actually stored in the array.
Notice the dtype, `Sparse[float64, nan]`. The `nan` means that elements in the array that are `nan` aren't actually stored, only the non-`nan` elements are. Those non-`nan` elements have a `float64` dtype.

The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA DataFrame:

```python
In [5]: df = pd.DataFrame(np.random.randn(10000, 4))
In [6]: df.iloc[:9998] = np.nan
In [7]: sdf = df.astype(pd.SparseDtype("float", np.nan))
In [8]: sdf.head()
Out[8]:
   0  1  2  3
0  NaN NaN NaN NaN
1  NaN NaN NaN NaN
2  NaN NaN NaN NaN
3  NaN NaN NaN NaN
4  NaN NaN NaN NaN
In [9]: sdf.dtypes
Out[9]:
0  Sparse[float64, nan]
1  Sparse[float64, nan]
2  Sparse[float64, nan]
3  Sparse[float64, nan]
dtype: object
In [10]: sdf.sparse.density
Out[10]: 0.0002
```

As you can see, the density (% of values that have not been “compressed”) is extremely low. This sparse object takes up much less memory on disk (pickled) and in the Python interpreter.

```python
In [11]: 'dense : {:0.2f} bytes'.format(df.memory_usage().sum() / 1e3)
Out[11]: 'dense : 320.13 bytes'
In [12]: 'sparse: {:0.2f} bytes'.format(sdf.memory_usage().sum() / 1e3)
Out[12]: 'sparse: 0.22 bytes'
```

Functionally, their behavior should be nearly identical to their dense counterparts.
### 2.25.1 SparseArray

`arrays.SparseArray` is an `ExtensionArray` for storing an array of sparse values (see `dtypes` for more on extension arrays). It is a 1-dimensional ndarray-like object storing only values distinct from the `fill_value`:

```python
In [13]: arr = np.random.randn(10)
In [14]: arr[2:5] = np.nan
In [15]: arr[7:8] = np.nan
In [16]: sparr = pd.arrays.SparseArray(arr)
In [17]: sparr
```
```
Out[17]:
[-1.9556635297215477, -1.6588664275960427, nan, nan, nan, 1.1589328886422277, 0.
 -1.4529711373305043, nan, 0.6060271905134522, 1.3342113401317768]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

A sparse array can be converted to a regular (dense) ndarray with `numpy.asarray()`

```python
In [18]: np.asarray(sparr)
```
```
Out[18]:
array([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
      nan, 0.606 , 1.3342])
```

### 2.25.2 SparseDtype

The `SparseArray.dtype` property stores two pieces of information

1. The dtype of the non-sparse values
2. The scalar fill value

```python
In [19]: sparr.dtype
Out[19]: Sparse[float64, nan]
```

A `SparseDtype` may be constructed by passing only a dtype

```python
In [20]: pd.SparseDtype(np.dtype('datetime64[ns]'))
```
```
Out[20]: Sparse[datetime64[ns], numpy.datetime64('NaT')]
```

in which case a default fill value will be used (for NumPy dtypes this is often the “missing” value for that dtype). To override this default an explicit fill value may be passed instead

```python
In [21]: pd.SparseDtype(np.dtype('datetime64[ns]'),
                 fill_value=pd.Timestamp('2017-01-01'))
```
```
Out[21]: Sparse[datetime64[ns], Timestamp('2017-01-01 00:00:00')]
```

Finally, the string alias 'Sparse[dttype]' may be used to specify a sparse dttype in many places
2.25.3 Sparse accessor

pandas provides a `.sparse` accessor, similar to `.str` for string data, `.cat` for categorical data, and `.dt` for datetime-like data. This namespace provides attributes and methods that are specific to sparse data.

In [23]: s = pd.Series([0, 0, 1, 2], dtype="Sparse[int"]")
In [24]: s.sparse.density
Out[24]: 0.5
In [25]: s.sparse.fill_value
Out[25]: 0

This accessor is available only on data with SparseDtype, and on the Series class itself for creating a Series with sparse data from a scipy COO matrix with.

New in version 0.25.0.

A `.sparse` accessor has been added for DataFrame as well. See Sparse accessor for more.

2.25.4 Sparse calculation

You can apply NumPy ufuncs to SparseArray and get a SparseArray as a result.

In [26]: arr = pd.arrays.SparseArray([1., np.nan, np.nan, -2., np.nan])
In [27]: np.abs(arr)
Out[27]:
[1.0, nan, nan, 2.0, nan]
Fill: nan
IntIndex
Indices: array([0, 3], dtype=int32)

The `ufunc` is also applied to `fill_value`. This is needed to get the correct dense result.

In [28]: arr = pd.arrays.SparseArray([1., -1, -1, -2., -1], fill_value=-1)
In [29]: np.abs(arr)
Out[29]:
[1, 1, 1, 2.0, 1]
Fill: 1
IntIndex
Indices: array([3], dtype=int32)

(continues on next page)
2.25.5 Migrating

Note: SparseSeries and SparseDataFrame were removed in pandas 1.0.0. This migration guide is present to aid in migrating from previous versions.

In older versions of pandas, the SparseSeries and SparseDataFrame classes (documented below) were the preferred way to work with sparse data. With the advent of extension arrays, these subclasses are no longer needed. Their purpose is better served by using a regular Series or DataFrame with sparse values instead.

Note: There’s no performance or memory penalty to using a Series or DataFrame with sparse values, rather than a SparseSeries or SparseDataFrame.

This section provides some guidance on migrating your code to the new style. As a reminder, you can use the Python warnings module to control warnings. But we recommend modifying your code, rather than ignoring the warning.

Construction

From an array-like, use the regular `Series` or `DataFrame` constructors with `SparseArray` values.

```
# Previous way
>>> pd.SparseDataFrame({"A": [0, 1]})

# New way
In [31]: pd.DataFrame({"A": pd.arrays.SparseArray([0, 1])})
Out[31]:
    A
0  0
1  1
```

From a SciPy sparse matrix, use `DataFrame.sparse.from_spmatrix()`.

```
# Previous way
>>> from scipy import sparse
>>> mat = sparse.eye(3)
>>> df = pd.SparseDataFrame(mat, columns=['A', 'B', 'C'])

# New way
In [32]: from scipy import sparse

In [33]: mat = sparse.eye(3)
In [34]: df = pd.DataFrame.sparse.from_spmatrix(mat, columns=['A', 'B', 'C'])
In [35]: df.dtypes
Out[35]:
A    Sparse[Float64, 0]
```

(continues on next page)
Conversion

From sparse to dense, use the `.sparse` accessors

```
In [36]: df.sparse.to_dense()
Out[36]:
     A  B  C
0  1.0 0.0 0.0
1  0.0 1.0 0.0
2  0.0 0.0 1.0
```

```
In [37]: df.sparse.to_coo()
Out[37]:
<3x3 sparse matrix of type '<class 'numpy.float64'>'
      with 3 stored elements in COOrdinate format>
```

From dense to sparse, use `DataFrame.astype()` with a `SparseDtype`.

```
In [38]: dense = pd.DataFrame({"A": [1, 0, 0, 1]})
In [39]: dtype = pd.SparseDtype(int, fill_value=0)
In [40]: dense.astype(dtype)
Out[40]:
     A
0  1
1  0
2  0
3  1
```

Sparse Properties

Sparse-specific properties, like density, are available on the `.sparse` accessor.

```
In [41]: df.sparse.density
Out[41]: 0.3333333333333333
```

General differences

In a `SparseDataFrame`, all columns were sparse. A `DataFrame` can have a mixture of sparse and dense columns. As a consequence, assigning new columns to a `DataFrame` with sparse values will not automatically convert the input to be sparse.

```python
# Previous Way
>>> df = pd.SparseDataFrame({"A": [0, 1]})
>>> df['B'] = [0, 0]  # implicitly becomes Sparse
>>> df['B'].dtype
Sparse[int64, nan]
```

Instead, you’ll need to ensure that the values being assigned are sparse.
```
In [42]: df = pd.DataFrame({"A": pd.arrays.SparseArray([0, 1])})

In [43]: df['B'] = [0, 0]  # remains dense

In [44]: df['B'].dtype
Out[44]: dtype('int64')

In [45]: df['B'] = pd.arrays.SparseArray([0, 0])

In [46]: df['B'].dtype
Out[46]: Sparse[int64, 0]
```

The SparseDataFrame.default_kind and SparseDataFrame.default_fill_value attributes have no replacement.

### 2.25.6 Interaction with scipy.sparse

Use `DataFrame.sparse.from_spmatrix()` to create a DataFrame with sparse values from a sparse matrix.

```
In [47]: from scipy.sparse import csr_matrix

In [48]: arr = np.random.random(size=(1000, 5))

In [49]: arr[arr < .9] = 0

In [50]: sp_arr = csr_matrix(arr)

In [51]: sp_arr
Out[51]: <1000x5 sparse matrix of type '<class 'numpy.float64'>'
      with 517 stored elements in Compressed Sparse Row format>

In [52]: sdf = pd.DataFrame.sparse.from_spmatrix(sp_arr)

In [53]: sdf.head()
Out[53]:
       0   1   2   3   4
0  0.956380 0.0  0.0  0.000000 0.0
1  0.000000 0.0  0.0  0.000000 0.0
2  0.000000 0.0  0.0  0.000000 0.0
3  0.000000 0.0  0.0  0.000000 0.0
4  0.999552 0.0  0.0  0.956153 0.0

In [54]: sdf.dtypes
Out[54]:
0  Sparse[float64, 0]
1  Sparse[float64, 0]
2  Sparse[float64, 0]
3  Sparse[float64, 0]
4  Sparse[float64, 0]
dtype: object
```
All sparse formats are supported, but matrices that are not in COOrdinate format will be converted, copying data as needed. To convert back to sparse SciPy matrix in COO format, you can use the `DataFrame.sparse.to_coo()` method:

In [55]: sdf.sparse.to_coo()
Out[55]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
    with 517 stored elements in COOrdinate format>

`Series.sparse.to_coo()` is implemented for transforming a Series with sparse values indexed by a `MultiIndex` to a scipy.sparse.coo_matrix.

The method requires a `MultiIndex` with two or more levels.

In [56]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [57]: s.index = pd.MultiIndex.from_tuples(
     ....:     [(1, 2, "a", 0),
     ....:      (1, 2, "a", 1),
     ....:      (1, 1, "b", 0),
     ....:      (1, 1, "b", 1),
     ....:      (2, 1, "b", 0),
     ....:      (2, 1, "b", 1),
     ....:     ],
     ....:     names=["A", "B", "C", "D"],
     ....: )

In [58]: ss = s.astype('Sparse')

In [59]: ss
Out[59]:
   A  B  C  D
  1  2  a  0  3.0
     1  NaN
  1  b  0  1.0
     1  3.0
  2  1  b  0  NaN
     1  NaN
dtype: Sparse[float64, nan]

In the example below, we transform the Series to a sparse representation of a 2-d array by specifying that the first and second `MultiIndex` levels define labels for the rows and the third and fourth levels define labels for the columns. We also specify that the column and row labels should be sorted in the final sparse representation.

In [60]: A, rows, columns = ss.sparse.to_coo(
     ....:     row_levels=['A', 'B'], column_levels=['C', 'D'], sort_labels=True
     ....: )

In [61]: A
Out[61]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'>

(continues on next page)
with 3 stored elements in COOrdinate format>

In [62]: A.todense()
Out[62]:
matrix([[0., 0., 1., 3.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])

In [63]: rows
Out[63]: [(1, 1), (1, 2), (2, 1)]

In [64]: columns
Out[64]: [('a', 0), ('a', 1), ('b', 0), ('b', 1)]

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

In [65]: A, rows, columns = ss.sparse.to_coo(
   ....: row_levels=['A', 'B', 'C'], column_levels=['D'], sort_labels=False
   ....: )
   ....:

In [66]: A
Out[66]:
<3x2 sparse matrix of type '<class 'numpy.float64'>'
  with 3 stored elements in COOrdinate format>

In [67]: A.todense()
Out[67]:
matrix([[3., 0.],
        [1., 3.],
        [0., 0.]])

In [68]: rows
Out[68]: [(1, 2, 'a'), (1, 1, 'b'), (2, 1, 'b')]

In [69]: columns
Out[69]: [(0,), (1,)]

A convenience method `Series.sparse.from_coo()` is implemented for creating a `Series` with sparse values from a scipy.sparse.coo_matrix.

In [70]: from scipy import sparse
In [71]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])), shape=(3, 4))
In [72]: A
Out[72]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
  with 3 stored elements in COOrdinate format>
In [73]: A.todense()
Out[73]:

matrix([[0., 0., 1., 2.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])

The default behaviour (with dense_index=False) simply returns a Series containing only the non-null entries.

In [74]: ss = pd.Series.sparse.from_coo(A)

In [75]: ss
Out[75]:
0  2    1.0
3  2.0
1  0    3.0
dtype: Sparse[float64, nan]

Specifying dense_index=True will result in an index that is the Cartesian product of the row and columns coordinates of the matrix. Note that this will consume a significant amount of memory (relative to dense_index=False) if the sparse matrix is large (and sparse) enough.

In [76]: ss_dense = pd.Series.sparse.from_coo(A, dense_index=True)

In [77]: ss_dense
Out[77]:
0  0    NaN
1   NaN
2  1.0
3  2.0
1  0    3.0
1   NaN
2   NaN
3   NaN
2  0    NaN
1   NaN
2   NaN
3   NaN
dtype: Sparse[float64, nan]

2.26 Frequently Asked Questions (FAQ)

2.26.1 DataFrame memory usage

The memory usage of a DataFrame (including the index) is shown when calling the info(). A configuration option, display.memory_usage (see the list of options), specifies if the DataFrame’s memory usage will be displayed when invoking the df.info() method.

For example, the memory usage of the DataFrame below is shown when calling info():

In [1]: dtypes = [
       ...:     "int64",
       ...:     "float64",
       ...:     "datetime64[ns]",
       ...: ]

(continues on next page)
...:        "timedelta64[ns]",
...:        "complex128",
...:        "object",
...:        "bool",
...:    ]
...

In [2]: n = 5000

In [3]: data = {t: np.random.randint(100, size=n).astype(t) for t in dtypes}

In [4]: df = pd.DataFrame(data)

In [5]: df["categorical"] = df["object"].astype("category")

In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
# Column    Non-Null Count  Dtype
---        --------------  -----  
    0 int64  5000 non-null  int64
    1 float64  5000 non-null  float64
    2 datetime64[ns]  5000 non-null  datetime64[ns]
    3 timedelta64[ns]  5000 non-null  timedelta64[ns]
    4 complex128  5000 non-null  complex128
    5 object  5000 non-null  object
    6 bool  5000 non-null  bool
    7 categorical  5000 non-null  category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
        object(1), timedelta64[ns](1)
memory usage: 288.2+ KB

The + symbol indicates that the true memory usage could be higher, because pandas does not count the memory used
by values in columns with dtype=object.

Passing memory_usage='deep' will enable a more accurate memory usage report, accounting for the full usage of
the contained objects. This is optional as it can be expensive to do this deeper introspection.

In [7]: df.info(memory_usage="deep")
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
# Column    Non-Null Count  Dtype
---        --------------  -----  
    0 int64  5000 non-null  int64
    1 float64  5000 non-null  float64
    2 datetime64[ns]  5000 non-null  datetime64[ns]
    3 timedelta64[ns]  5000 non-null  timedelta64[ns]
    4 complex128  5000 non-null  complex128
    5 object  5000 non-null  object
    6 bool  5000 non-null  bool
    7 categorical  5000 non-null  category

(continues on next page)
By default the display option is set to `True` but can be explicitly overridden by passing the `memory_usage` argument when invoking `df.info()`.

The memory usage of each column can be found by calling the `memory_usage()` method. This returns a `Series` with an index represented by column names and memory usage of each column shown in bytes. For the `DataFrame` above, the memory usage of each column and the total memory usage can be found with the `memory_usage` method:

```python
In [8]: df.memory_usage()
Out[8]:
       Index     128
      int64  40000
      float64  40000
     datetime64[ns]  40000
     timedelta64[ns]  40000
        object  40000
          bool    5000
     categorical     9968
dtype: int64

# total memory usage of dataframe
In [9]: df.memory_usage().sum()
Out[9]: 295096
```

By default the memory usage of the `DataFrame`’s index is shown in the returned `Series`, the memory usage of the index can be suppressed by passing the `index=False` argument:

```python
In [10]: df.memory_usage(index=False)
Out[10]:
       int64   40000
      float64   40000
     datetime64[ns]   40000
     timedelta64[ns]   40000
        complex128   80000
          object   40000
          bool    5000
     categorical     9968
dtype: int64
```

The memory usage displayed by the `info()` method utilizes the `memory_usage()` method to determine the memory usage of a `DataFrame` while also formatting the output in human-readable units (base-2 representation; i.e. 1KB = 1024 bytes).

See also `Categorical Memory Usage`. 
2.26.2 Using if/truth statements with pandas

pandas follows the NumPy convention of raising an error when you try to convert something to a bool. This happens in an if-statement or when using the boolean operations: and, or, and not. It is not clear what the result of the following code should be:

```python
>>> if pd.Series([False, True, False]):
...   pass
```

Should it be True because it’s not zero-length, or False because there are False values? It is unclear, so instead, pandas raises a ValueError:

```python
>>> if pd.Series([False, True, False]):
...   print("I was true")
Traceback
...   ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

You need to explicitly choose what you want to do with the DataFrame, e.g. use any(), all() or empty(). Alternatively, you might want to compare if the pandas object is None:

```python
>>> if pd.Series([False, True, False]) is not None:
...   print("I was not None")
I was not None
```

Below is how to check if any of the values are True:

```python
>>> if pd.Series([False, True, False]).any():
...   print("I am any")
I am any
```

To evaluate single-element pandas objects in a boolean context, use the method bool():

```python
In [11]: pd.Series([True]).bool()
Out[11]: True

In [12]: pd.Series([False]).bool()
Out[12]: False

In [13]: pd.DataFrame([[True]]).bool()
Out[13]: True

In [14]: pd.DataFrame([[False]]).bool()
Out[14]: False
```
Bitwise boolean

Bitwise boolean operators like == and != return a boolean Series, which is almost always what you want anyways.

```python
>>> s = pd.Series(range(5))
>>> s == 4
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

See [boolean comparisons](#) for more examples.

Using the in operator

Using the Python in operator on a Series tests for membership in the index, not membership among the values.

```python
In [15]: s = pd.Series(range(5), index=list("abcde"))

In [16]: 2 in s
Out[16]: False

In [17]: 'b' in s
Out[17]: True
```

If this behavior is surprising, keep in mind that using in on a Python dictionary tests keys, not values, and Series are dict-like. To test for membership in the values, use the method isin():

```python
In [18]: s.isin([2])
Out[18]:
a   False
b   False
c   True
d   False
e   False
dtype: bool

In [19]: s.isin([2]).any()
Out[19]: True
```

For DataFrames, likewise, in applies to the column axis, testing for membership in the list of column names.
2.26.3 Mutating with User Defined Function (UDF) methods

This section applies to pandas methods that take a UDF. In particular, the methods `.apply`, `.aggregate`, `.transform`, and `.filter`.

It is a general rule in programming that one should not mutate a container while it is being iterated over. Mutation will invalidate the iterator, causing unexpected behavior. Consider the example:

```
In [20]: values = [0, 1, 2, 3, 4, 5]
In [21]: n_removed = 0
In [22]: for k, value in enumerate(values):
    ...:     idx = k - n_removed
    ...:     if value % 2 == 1:
    ...:         del values[idx]
    ...:         n_removed += 1
    ...:     else:
    ...:         values[idx] = value + 1

In [23]: values
Out[23]: [1, 4, 5]
```

One probably would have expected that the result would be [1, 3, 5]. When using a pandas method that takes a UDF, internally pandas is often iterating over the DataFrame or other pandas object. Therefore, if the UDF mutates (changes) the DataFrame, unexpected behavior can arise.

Here is a similar example with `DataFrame.apply()`:

```
In [24]: def f(s):
    ...:     s.pop("a")
    ...:     return s

In [25]: df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
In [26]: try:
    ...:     df.apply(f, axis="columns")
    ...: except Exception as err:
    ...:     print(repr(err))
    ...: KeyException('a')
```

To resolve this issue, one can make a copy so that the mutation does not apply to the container being iterated over.

```
In [27]: values = [0, 1, 2, 3, 4, 5]
In [28]: n_removed = 0
In [29]: for k, value in enumerate(values.copy()):
    ...:     idx = k - n_removed
    ...:     if value % 2 == 1:
    ...:         del values[idx]
    ...:         n_removed += 1
```
In [30]: values
Out[30]: [1, 3, 5]

In [31]: def f(s):
    ....:     s = s.copy()
    ....:     s.pop("a")
    ....:     return s
In [32]: df = pd.DataFrame({"a": [1, 2, 3], 'b': [4, 5, 6]})
In [33]: df.apply(f, axis="columns")
Out[33]:
    b
0  4
1  5
2  6

2.26.4 NaN, Integer NA values and NA type promotions

Choice of NA representation

For lack of NA (missing) support from the ground up in NumPy and Python in general, we were given the difficult choice between either:

- A masked array solution: an array of data and an array of boolean values indicating whether a value is there or is missing.

- Using a special sentinel value, bit pattern, or set of sentinel values to denote NA across the dtypes.

For many reasons we chose the latter. After years of production use it has proven, at least in my opinion, to be the best decision given the state of affairs in NumPy and Python in general. The special value NaN (Not-A-Number) is used everywhere as the NA value, and there are API functions isna and notna which can be used across the dtypes to detect NA values.

However, it comes with it a couple of trade-offs which I most certainly have not ignored.

Support for integer NA

In the absence of high performance NA support being built into NumPy from the ground up, the primary casualty is the ability to represent NAs in integer arrays. For example:

In [34]: s = pd.Series([1, 2, 3, 4, 5], index=list("abcde"))
In [35]: s
Out[35]:
   a    1
   b    2
(continues on next page)
c 3
d 4
e 5
dtype: int64

In [36]: s.dtype
Out[36]: dtype('int64')

In [37]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [38]: s2
Out[38]:
a  1.0
b  2.0
c  3.0
f  NaN
u  NaN
dtype: float64

In [39]: s2.dtype
Out[39]: dtype('float64')

This trade-off is made largely for memory and performance reasons, and also so that the resulting Series continues to be “numeric”.

If you need to represent integers with possibly missing values, use one of the nullable-integer extension dtypes provided by pandas:

- Int8Dtype
- Int16Dtype
- Int32Dtype
- Int64Dtype

In [40]: s_int = pd.Series([1, 2, 3, 4, 5], index=list('abcde'), dtype=pd.Int64Dtype())

In [41]: s_int
Out[41]:
a  1
b  2
c  3
d  4
e  5
dtype: Int64

In [42]: s_int.dtype
Out[42]: Int64Dtype()

In [43]: s2_int = s_int.reindex(['a', 'b', 'c', 'f', 'u'])

In [44]: s2_int
Out[44]:
a  1
(continues on next page)
See *Nullable integer data type* for more.

**NA type promotions**

When introducing NAs into an existing *Series* or *DataFrame* via `reindex()` or some other means, boolean and integer types will be promoted to a different dtype in order to store the NAs. The promotions are summarized in this table:

<table>
<thead>
<tr>
<th>Typeclass</th>
<th>Promotion dtype for storing NAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating</td>
<td>no change</td>
</tr>
<tr>
<td>object</td>
<td>no change</td>
</tr>
<tr>
<td>integer</td>
<td>cast to float64</td>
</tr>
<tr>
<td>boolean</td>
<td>cast to object</td>
</tr>
</tbody>
</table>

While this may seem like a heavy trade-off, I have found very few cases where this is an issue in practice i.e. storing values greater than $2^{**53}$. Some explanation for the motivation is in the next section.

**Why not make NumPy like R?**

Many people have suggested that NumPy should simply emulate the NA support present in the more domain-specific statistical programming language R. Part of the reason is the NumPy type hierarchy:

<table>
<thead>
<tr>
<th>Typeclass</th>
<th>Dtypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.floating</td>
<td>float16, float32, float64, float128</td>
</tr>
<tr>
<td>numpy.integer</td>
<td>int8, int16, int32, int64</td>
</tr>
<tr>
<td>numpy.unsignedinteger</td>
<td>uint8, uint16, uint32, uint64</td>
</tr>
<tr>
<td>numpy.object_</td>
<td>object_</td>
</tr>
<tr>
<td>numpy.bool_</td>
<td>bool_</td>
</tr>
<tr>
<td>numpy.character</td>
<td>string_, unicode_</td>
</tr>
</tbody>
</table>

The R language, by contrast, only has a handful of built-in data types: integer, numeric (floating-point), character, and boolean. NA types are implemented by reserving special bit patterns for each type to be used as the missing value. While doing this with the full NumPy type hierarchy would be possible, it would be a more substantial trade-off (especially for the 8- and 16-bit data types) and implementation undertaking.

An alternate approach is that of using masked arrays. A masked array is an array of data with an associated boolean *mask* denoting whether each value should be considered NA or not. I am personally not in love with this approach as I feel that overall it places a fairly heavy burden on the user and the library implementer. Additionally, it exacts a fairly high performance cost when working with numerical data compared with the simple approach of using NaN. Thus, I have chosen the Pythonic "practicality beats purity" approach and traded integer NA capability for a much simpler approach of using a special value in float and object arrays to denote NA, and promoting integer arrays to floating when NAs must be introduced.
2.26.5 Differences with NumPy

For Series and DataFrame objects, \texttt{var()} normalizes by \(N-1\) to produce unbiased estimates of the sample variance, while NumPy's \texttt{var} normalizes by \(N\), which measures the variance of the sample. Note that \texttt{cov()} normalizes by \(N-1\) in both pandas and NumPy.

2.26.6 Thread-safety

As of pandas 0.11, pandas is not 100\% thread safe. The known issues relate to the \texttt{copy()} method. If you are doing a lot of copying of DataFrame objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See this link for more information.

2.26.7 Byte-ordering issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. A common symptom of this issue is an error like:

```
Traceback
... ValueError: Big-endian buffer not supported on little-endian compiler
```

To deal with this issue you should convert the underlying NumPy array to the native system byte order before passing it to Series or DataFrame constructors using something similar to the following:

```
In [46]: x = np.array(list(range(10)),">i4") # big endian
In [47]: newx = x.byteswap().newbyteorder() # force native byteorder
In [48]: s = pd.Series(newx)
```

See the NumPy documentation on byte order for more details.

2.27 Cookbook

This is a repository for short and sweet examples and links for useful pandas recipes. We encourage users to add to this documentation.

Adding interesting links and/or inline examples to this section is a great First Pull Request.

Simplified, condensed, new-user friendly, in-line examples have been inserted where possible to augment the Stack-Overflow and GitHub links. Many of the links contain expanded information, above what the in-line examples offer.

pandas (pd) and NumPy (np) are the only two abbreviated imported modules. The rest are kept explicitly imported for newer users.
2.27.1 Idioms

These are some neat pandas idioms

if-then/if-then-else on one column, and assignment to another one or more columns:

```
In [1]: df = pd.DataFrame(
    ...:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
    ...: )
    ...

In [2]: df
Out[2]:
      AAA  BBB  CCC
0     4   10  100
1     5    20   50
2     6    30  -30
3     7    40  -50
```

if-then...

An if-then on one column

```
In [3]: df.loc[df.AAA >= 5, "BBB"] = -1

In [4]: df
Out[4]:
      AAA  BBB  CCC
0     4   10  100
1     5   -1   50
2     6   -1  -30
3     7   -1  -50
```

An if-then with assignment to 2 columns:

```
In [5]: df.loc[df.AAA >= 5, ["BBB", "CCC"]]] = 555

In [6]: df
Out[6]:
      AAA  BBB  CCC
0     4   10  100
1     5  555  555
2     6  555  555
3     7  555  555
```

Add another line with different logic, to do the -else

```
In [7]: df.loc[df.AAA < 5, ["BBB", "CCC"]]] = 2000

In [8]: df
Out[8]:
      AAA  BBB  CCC
0     4  2000  2000
```
(continues on next page)
Or use pandas where after you’ve set up a mask

```
In [9]: df_mask = pd.DataFrame(
    ...:     {"AAA": [True] * 4, "BBB": [False] * 4, "CCC": [True, False] * 2}
    ...: )
    ...

In [10]: df.where(df_mask, -1000)
Out[10]:
   AAA  BBB  CCC
0   4  1000  100
1   5 -1000 -100
2   6  555  -100
3   7 -1000 -1000
```

if-then-else using NumPy’s where()

```
In [11]: df = pd.DataFrame(
    ...:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
    ...: )
    ...

In [12]: df
Out[12]:
   AAA  BBB  CCC
0   4   10   100
1   5   20    50
2   6   30   -30
3   7   40   -50

In [13]: df["logic"] = np.where(df["AAA"] > 5, "high", "low")
In [14]: df
Out[14]:
   AAA  BBB  CCC  logic
0   4   10  100   low
1   5   20   50   low
2   6   30  -30  high
3   7   40  -50  high
Splitting

Split a frame with a boolean criterion

```
In [15]: df = pd.DataFrame(
    ....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
    ....: )
    ....:

In [16]: df
Out[16]:
   AAA  BBB  CCC
0   4   10  100
1   5   20   50
2   6   30  -30
3   7   40  -50

In [17]: df[df.AAA <= 5]
Out[17]:
   AAA  BBB  CCC
0   4   10  100
1   5   20   50

In [18]: df[df.AAA > 5]
Out[18]:
   AAA  BBB  CCC
2   6   30  -30
3   7   40  -50
```

Building criteria

Select with multi-column criteria

```
In [19]: df = pd.DataFrame(
    ....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
    ....: )
    ....:

In [20]: df
Out[20]:
   AAA  BBB  CCC
0   4   10  100
1   5   20   50
2   6   30  -30
3   7   40  -50

In [21]: df.loc[(df["BBB"] < 25) & (df["CCC"] >= -40), "AAA"]
Out[21]:
0   4
1   5
Name: AAA, dtype: int64
```

...or (without assignment returns a Series)

```
In [22]: df.loc[(df["BBB"] > 25) | (df["CCC"] >= -40), "AAA"]
Out[22]:
   AAA
0   4
1   5
2   6
3   7
Name: AAA, dtype: int64
```

...or (with assignment modifies the DataFrame.)

```
In [23]: df.loc[(df["BBB"] > 25) | (df["CCC"] >= 75), "AAA"] = 0.1
In [24]: df
Out[24]:
   AAA  BBB  CCC
0   0.1  10   100
1   5.0  20    50
2  0.1  30   -30
3  0.1  40   -50
```

Select rows with data closest to certain value using argsort

```
In [25]: df = pd.DataFrame(
   ....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
   ....: )
   ....:
In [26]: df
Out[26]:
   AAA  BBB  CCC
0   4  10  100
1   5  20   50
2   6  30  -30
3   7  40  -50
In [27]: aValue = 43.0
In [28]: df.loc[(df.CCC - aValue).abs().argsort()]
Out[28]:
   AAA  BBB  CCC
1   5  20   50
0   4  10  100
2   6  30  -30
3   7  40  -50
```

Dynamically reduce a list of criteria using a binary operators

```
In [29]: df = pd.DataFrame(
   ....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
   ....: )
   ....:
```

(continues on next page)
In [30]: df
Out[30]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>

In [31]: Crit1 = df.AAA <= 5.5
In [32]: Crit2 = df.BBB == 10.0
In [33]: Crit3 = df.CCC > -40.0

One could hard code:
In [34]: AllCrit = Crit1 & Crit2 & Crit3

...Or it can be done with a list of dynamically built criteria
In [35]: import functools
In [36]: CritList = [Crit1, Crit2, Crit3]
In [37]: AllCrit = functools.reduce(lambda x, y: x & y, CritList)
In [38]: df[AllCrit]
Out[38]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

2.27.2 Selection

Dataframes

The indexing docs.

Using both row labels and value conditionals
In [39]: df = pd.DataFrame(
    .....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
    .....:     )
In [40]: df
Out[40]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>
In [41]: df[(df.AAA <= 6) & (df.index.isin([0, 2, 4]))]

Out[41]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

Use `loc` for label-oriented slicing and `iloc` positional slicing GH2904

In [42]: df = pd.DataFrame(
      ...:
      ....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]},
      ...:
      ....:     index=["foo", "bar", "boo", "kar"],
      ...:
      ....: )

There are 2 explicit slicing methods, with a third general case

1. Positional-oriented (Python slicing style: exclusive of end)
2. Label-oriented (Non-Python slicing style: inclusive of end)
3. General (Either slicing style: depends on if the slice contains labels or positions)

In [43]: df.loc["bar":"kar"]  # Label

Out[43]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>boo</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>kar</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>

# Generic

In [44]: df[0:3]

Out[44]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>bar</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>boo</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

In [45]: df["bar":"kar"]

Out[45]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>boo</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>kar</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>

Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.

In [46]: data = {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}

In [47]: df2 = pd.DataFrame(data=data, index=[1, 2, 3, 4])  # Note index starts at 1.

In [48]: df2.iloc[1:3]  # Position-oriented

Out[48]:
<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>bar</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>boo</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>
In [49]: df2.loc[1:3]  # Label-oriented
Out[49]:

<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

Using inverse operator (\~) to take the complement of a mask

In [50]: df = pd.DataFrame(
   ....:   {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
   ....: )
   ....:
In [51]: df
Out[51]:

<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>

In [52]: df[~((df.AAA <= 6) & (df.index.isin([0, 2, 4])))]
Out[52]:

<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>40</td>
</tr>
</tbody>
</table>

New columns

Efficiently and dynamically creating new columns using applymap

In [53]: df = pd.DataFrame({"AAA": [1, 2, 1, 3], "BBB": [1, 1, 2, 2], "CCC": [2, 1, 3, 1]})
In [54]: df
Out[54]:

<table>
<thead>
<tr>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

In [55]: source_cols = df.columns  # Or some subset would work too
In [56]: new_cols = [str(x) + "_cat" for x in source_cols]
In [57]: categories = {1: "Alpha", 2: "Beta", 3: "Charlie"}

In [58]: df[new_cols] = df[source_cols].applymap(categories.get)

In [59]: df

Out[59]:

<table>
<thead>
<tr>
<th></th>
<th>AAA</th>
<th>BBB</th>
<th>CCC</th>
<th>AAA_cat</th>
<th>BBB_cat</th>
<th>CCC_cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>Alpha</td>
<td>Alpha</td>
<td>Beta</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Beta</td>
<td>Alpha</td>
<td>Alpha</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Alpha</td>
<td>Beta</td>
<td>Charlie</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Charlie</td>
<td>Beta</td>
<td>Alpha</td>
</tr>
</tbody>
</table>

Keep other columns when using min() with groupby

In [60]: df = pd.DataFrame(
        .....:   {"AAA": [1, 1, 1, 2, 2, 2, 3, 3], "BBB": [2, 1, 3, 4, 5, 1, 2, 3]}
        .....: )
        .....:

In [61]: df

Out[61]:

<table>
<thead>
<tr>
<th></th>
<th>AAA</th>
<th>BBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Method 1: idxmin() to get the index of the minimums

In [62]: df.loc[df.groupby("AAA")["BBB"].idxmin()]

Out[62]:

<table>
<thead>
<tr>
<th></th>
<th>AAA</th>
<th>BBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Method 2: sort then take first of each

In [63]: df.sort_values(by="BBB").groupby("AAA", as_index=False).first()

Out[63]:

<table>
<thead>
<tr>
<th></th>
<th>AAA</th>
<th>BBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Notice the same results, with the exception of the index.
2.27.3 Multiindexing

The *multindexing* docs.

Creating a MultiIndex from a labeled frame

```python
In [64]: df = pd.DataFrame(
    ....:     {
    ....:         "row": [0, 1, 2],
    ....:         "One_X": [1.1, 1.1, 1.1],
    ....:         "One_Y": [1.2, 1.2, 1.2],
    ....:         "Two_X": [1.11, 1.11, 1.11],
    ....:         "Two_Y": [1.22, 1.22, 1.22],
    ....:     }
    ....: )
    ....:
In [65]: df
Out[65]:
<table>
<thead>
<tr>
<th>row</th>
<th>One_X</th>
<th>One_Y</th>
<th>Two_X</th>
<th>Two_Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
<td>1.22</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
<td>1.22</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
<td>1.22</td>
</tr>
</tbody>
</table>

# As Labelled Index
In [66]: df = df.set_index("row")
```

```python
In [67]: df
Out[67]:
<table>
<thead>
<tr>
<th>One_X</th>
<th>One_Y</th>
<th>Two_X</th>
<th>Two_Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>1.2</td>
<td>1.11</td>
</tr>
</tbody>
</table>
```

# With Hierarchical Columns
In [68]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split("_"))) for c in df.columns])
```

```python
In [69]: df
Out[69]:
<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>row</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1.1</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
</tr>
</tbody>
</table>

# Now stack & Reset
In [70]: df = df.stack(0).reset_index(1)
```

```python
In [71]: df
Out[71]:
<table>
<thead>
<tr>
<th>level_1</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
# And fix the labels (Notice the label 'level_1' got added automatically)

```python
In [72]: df.columns = ['Sample', 'All_X', 'All_Y']
```

```python
In [73]: df
```

```
Sample All_X All_Y
row
0 One 1.10 1.20
0 Two 1.11 1.22
1 One 1.10 1.20
1 Two 1.11 1.22
2 One 1.10 1.20
2 Two 1.11 1.22
```

## Arithmetic

Performing arithmetic with a MultiIndex that needs broadcasting

```python
In [74]: cols = pd.MultiIndex.from_tuples((
   ...:     (x, y) for x in ['A', 'B', 'C'] for y in ['O', 'I'])
   ...: )
   ...
```

```python
In [75]: df = pd.DataFrame(np.random.randn(2, 6), index=['n', 'm'], columns=cols)
```

```python
In [76]: df
```

```
A B C
n 0.469112 -0.282863 -1.509059 -1.135632 1.212112 -0.173215
m 0.119209 -1.044236 -0.861849 -2.104569 -0.494929 1.071804
```

```python
In [77]: df = df.div(df['C'], level=1)
```

```python
In [78]: df
```

```
A B C
n 0.387021 1.633022 -1.244983 6.556214 1.0 1.0
m -0.240860 -0.974279 1.741358 -1.963577 1.0 1.0
```
Slicing

Slicing a MultiIndex with `xs`

```
In [79]: coords = ["AA", "one"], ["AA", "six"], ["BB", "one"], ["BB", "two"], ["BB", "six"]

In [80]: index = pd.MultiIndex.from_tuples(coords)

In [81]: df = pd.DataFrame([11, 22, 33, 44, 55], index, ["MyData"])

In [82]: df
Out[82]:
   MyData
AA one   11
     six  22
BB one   33
     two  44
     six  55

To take the cross section of the 1st level and 1st axis the index:

```
In [83]: df.xs("BB", level=0, axis=0)
Out[83]:
   MyData
one 33
two 44
six 55
```

...and now the 2nd level of the 1st axis.

```
In [84]: df.xs("six", level=1, axis=0)
Out[84]:
   MyData
AA 22
BB 55
```

Slicing a MultiIndex with `xs`, method #2

```
In [85]: import itertools

In [86]: index = list(itertools.product(["Ada", "Quinn", "Violet"], ["Comp", "Math", "Sci"]))

In [87]: headr = list(itertools.product(["Exams", "Labs"], ["I", "II"]))

In [88]: indx = pd.MultiIndex.from_tuples(index, names=["Student", "Course"])

In [89]: cols = pd.MultiIndex.from_tuples(headr)  # Notice these are un-named

In [90]: data = [70 + x + y + (x * y) % 3 for x in range(4) for y in range(9)]

In [91]: df = pd.DataFrame(data, indx, cols)
```

(continues on next page)
In [92]: df
Out[92]:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Exams</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Comp</td>
<td>70 71</td>
<td>72 73</td>
</tr>
<tr>
<td>Ada</td>
<td>Math</td>
<td>71 73</td>
<td>75 74</td>
</tr>
<tr>
<td>Quinn</td>
<td>Comp</td>
<td>73 74</td>
<td>75 76</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>74 76</td>
<td>78 77</td>
</tr>
<tr>
<td>Violet</td>
<td>Comp</td>
<td>76 77</td>
<td>78 79</td>
</tr>
<tr>
<td>Violet</td>
<td>Math</td>
<td>77 79</td>
<td>81 80</td>
</tr>
<tr>
<td>Violet</td>
<td>Sci</td>
<td>78 81</td>
<td>81 81</td>
</tr>
</tbody>
</table>

In [93]: All = slice(None)

In [94]: df.loc["Violet"]
Out[94]:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Exams</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Math</td>
<td>71 73</td>
<td>75 74</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>74 76</td>
<td>78 77</td>
</tr>
<tr>
<td>Violet</td>
<td>Math</td>
<td>77 79</td>
<td>81 80</td>
</tr>
</tbody>
</table>

In [95]: df.loc[(All, "Math"), All]
Out[95]:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Exams</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Math</td>
<td>71 73</td>
<td>75 74</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>74 76</td>
<td>78 77</td>
</tr>
<tr>
<td>Violet</td>
<td>Math</td>
<td>77 79</td>
<td>81 80</td>
</tr>
</tbody>
</table>

In [96]: df.loc[(slice("Ada", "Quinn"), "Math"), All]
Out[96]:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Exams</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Math</td>
<td>71 73</td>
<td>75 74</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>74 76</td>
<td>78 77</td>
</tr>
</tbody>
</table>

In [97]: df.loc[(All, "Math"), ("Exams")]
Out[97]:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Exams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Math</td>
<td>71 73</td>
</tr>
<tr>
<td>Quinn</td>
<td>Math</td>
<td>74 76</td>
</tr>
<tr>
<td>Violet</td>
<td>Math</td>
<td>77 79</td>
</tr>
</tbody>
</table>
Setting portions of a MultiIndex with `xs`:

Sorting:

Sort by specific column or an ordered list of columns, with a MultiIndex:

Partial selection, the need for sortedness GH2995:

Levels:

Prepending a level to a multiindex:

Flatten Hierarchical columns:

2.27.4 Missing data:

The `missing data` docs.

Fill forward a reversed timeseries:

(continues on next page)
In [101]: df.loc[df.index[3], "A"] = np.nan

In [102]: df
Out[102]:
   A
0  2013-08-01 0.721555
1  2013-08-02 -0.706771
2  2013-08-05 -1.039575
3  2013-08-06 NaN
4  2013-08-07 -0.424972
5  2013-08-08  0.567020

In [103]: df.reindex(df.index[::-1]).ffill()
Out[103]:
   A
0  2013-08-08 0.567020
1  2013-08-07 -0.424972
2  2013-08-06 -0.424972
3  2013-08-05 -1.039575
4  2013-08-02 -0.706771
5  2013-08-01  0.721555

cumsum reset at NaN values

Replace

Using replace with backrefs

2.27.5 Grouping

The grouping docs.

Basic grouping with apply

Unlike agg, apply’s callable is passed a sub-DataFrame which gives you access to all the columns

In [104]: df = pd.DataFrame({
    .....:     "animal": "cat dog cat fish dog cat".split(),
    .....:     "size": list("SSMMMLL"),
    .....:     "weight": [8, 10, 11, 1, 20, 12, 12],
    .....:     "adult": [False] * 5 + [True] * 2,
    .....:   )
    .....: })

In [105]: df
Out[105]:
   animal  size  weight  adult
0  cat     S      8     False
1  dog     S     10     False
2  cat     M     11     False

(continues on next page)
# List the size of the animals with the highest weight.

```python
In [106]: df.groupby("animal").apply(lambda subf: subf["size"][subf["weight"].idxmax()])
Out[106]:
animal
    cat  L
    dog  M
    fish M
dtype: object
```

Using `get_group`

```python
In [107]: gb = df.groupby(['animal'])
In [108]: gb.get_group("cat")
Out[108]:
animal size  weight  adult
   0     cat   S  8.00 False
   2     cat   M 11.00 False
   5     cat   L 12.00  True
   6     cat   L 12.00  True
```

Apply to different items in a group

```python
In [109]: def GrowUp(x):
    ....:     avg_weight = sum(x[x["size"] == "S"].weight * 1.5)
    ....:     avg_weight += sum(x[x["size"] == "M"].weight * 1.25)
    ....:     avg_weight += sum(x[x["size"] == "L"].weight)
    ....:     avg_weight /= len(x)
    ....:     return pd.Series(["L", avg_weight, True], index=["size", "weight", "adult"])
    ....:
In [110]: expected_df = gb.apply(GrowUp)
In [111]: expected_df
```

```
size  weight  adult
animal
    cat  L 12.44  True
    dog  L 20.00  True
    fish L 1.25  True
```

Expanding apply

```python
In [112]: S = pd.Series([i / 100.0 for i in range(1, 11)])
In [113]: def cum_ret(x, y):
    ....:     return x * (1 + y)
    ....:
```
In [114]: def red(x):
   ...:     return functools.reduce(cum_ret, x, 1.0)
   ...:

In [115]: S.expanding().apply(red, raw=True)
Out[115]:
   0    1.010000
   1    1.030200
   2    1.061106
   3    1.103550
   4    1.158728
   5    1.228251
   6    1.314229
   7    1.419367
   8    1.547110
   9    1.701821
   dtype: float64

Replacing some values with mean of the rest of a group

In [116]: df = pd.DataFrame({"A": [1, 1, 2, 2], "B": [1, -1, 1, 2]})
In [117]: gb = df.groupby("A")
In [118]: def replace(g):
   ...:     mask = g < 0
   ...:     return g.where(mask, g[~mask].mean())
   ...:

In [119]: gb.transform(replace)
Out[119]:
   B
   0  1.0
   1 -1.0
   2  1.5
   3  1.5

gb.transform(replace)

Sort groups by aggregated data

In [120]: df = pd.DataFrame(
   ...:     { "code": ["foo", "bar", "baz"] * 2,
   ...:     "data": [0.16, -0.21, 0.33, 0.45, -0.59, 0.62],
   ...:     "flag": [False, True] * 3,
   ...:     }
   ...
   ...
)

In [121]: code_groups = df.groupby("code")
In [122]: agg_n_sort_order = code_groups["data"].transform(sum).sort_values(by="data")

In [123]: sorted_df = df.loc[agg_n_sort_order.index]

In [124]: sorted_df

Out[124]:
  code  data  flag
0  foo  0.16  False
1  bar -0.21  True
2  baz  0.33  False
3  foo  0.45  True
4  bar -0.59  False
5  baz  0.62  True

Create multiple aggregated columns

In [125]: rng = pd.date_range(start="2014-10-07", periods=10, freq="2min")

In [126]: ts = pd.Series(data=list(range(10)), index=rng)

In [127]: def MyCust(x):
      .....:     if len(x) > 2:
      .....:         return x[1] * 1.234
      .....:     return pd.NaT

In [128]: mhc = {"Mean": np.mean, "Max": np.max, "Custom": MyCust}

In [129]: ts.resample("5min").apply(mhc)

Out[129]:
               Mean   Max  Custom
2014-10-07 00:00:00  1.0   2  1.234
2014-10-07 00:05:00  3.5   4    NaT
2014-10-07 00:10:00  6.0   7  7.404
2014-10-07 00:15:00  8.5   9    NaT

In [130]: ts

Out[130]:
2014-10-07 00:00:00    0
2014-10-07 00:02:00    1
2014-10-07 00:04:00    2
2014-10-07 00:06:00    3
2014-10-07 00:08:00    4
2014-10-07 00:10:00    5
2014-10-07 00:12:00    6
2014-10-07 00:14:00    7
2014-10-07 00:16:00    8
2014-10-07 00:18:00    9
Freq: 2T, dtype: int64

Create a value counts column and reassign back to the DataFrame
In [131]: df = pd.DataFrame(
    ....:     {"Color": "Red Red Red Blue".split(), "Value": [100, 150, 50, 50]}
    ....: )
    ....:
In [132]: df
Out[132]:
   Color  Value
0   Red   100
1   Red   150
2   Red    50
3   Blue    50

In [133]: df["Counts"] = df.groupby(["Color"]).transform(len)

In [134]: df
Out[134]:
   Color  Value  Counts
0   Red   100     3
1   Red   150     3
2   Red    50     3
3   Blue    50     1

Shift groups of the values in a column based on the index

In [135]: df = pd.DataFrame(
    ....:     {"line_race": [10, 10, 8, 10, 10, 8], "beyer": [99, 102, 103, 103, 88, 100]},
    ....:     index=[
    ....:         "Last Gunfighter",
    ....:         "Last Gunfighter",
    ....:         "Last Gunfighter",
    ....:         "Paynter",
    ....:         "Paynter",
    ....:         "Paynter"],
    ....: )
    ....:
In [136]: df
Out[136]:
   line_race  beyer
Last Gunfighter    10   99
Last Gunfighter    10  102
Last Gunfighter     8  103
Paynter            10  103
Paynter            10   88
Paynter             8  100

In [137]: df["beyer_shifted"] = df.groupby(level=0)["beyer"]).shift(1)

In [138]: df
Out[138]: (continues on next page)
Select row with maximum value from each group

```python
In [139]: df = pd.DataFrame(
    ......:     {
    ......:         "host": ["other", "other", "that", "this", "this"],
    ......:         "service": ["mail", "web", "mail", "mail", "web"],
    ......:         "no": [1, 2, 1, 2, 1],
    ......:     }
    ......: ).set_index(["host", "service"])
    ......:
In [140]: mask = df.groupby(level=0).agg("idxmax")
In [141]: df_count = df.loc[mask["no"]].reset_index()
In [142]: df_count
Out[142]:
   host  service  no
0   other    web  2
1    that    mail 1
2    this    mail 2
```

Grouping like Python’s itertools.groupby

```python
In [143]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=["A"])
In [144]: df["A"].groupby((df["A"] != df["A"].shift()).cumsum()).groups
Out[144]: {1: [0], 2: [1], 3: [2], 4: [3, 4, 5], 5: [6], 6: [7, 8]}
In [145]: df["A"].groupby((df["A"] != df["A"].shift()).cumsum()).cumsum()
Out[145]:
   0   1
0   0   1
1   0   1
2   0   1
3   1   1
4   2   1
5   3   1
6   0   1
7   1   1
8   2   1
Name: A, dtype: int64
```
Expanding data

Alignment and to-date
Rolling Computation window based on values instead of counts
Rolling Mean by Time Interval

Splitting

Splitting a frame
Create a list of dataframes, split using a delineation based on logic included in rows.

```
In [146]: df = pd.DataFrame(
   .....:     data={
   .....:         "Data": np.random.randn(9),
   .....:     }
   .....: )
   .....:
In [147]: dfs = list(
   .....:     zip(*df.groupby(
   .....:         1 * (df["Case"] == "B"))
   .....:         .cumsum()
   .....:         .rolling(window=3, min_periods=1)
   .....:         .median())
   .....: )[-1]
   .....:
In [148]: dfs[0]
Out[148]:
   Case  Data
0     A  0.276232
1     A -1.087401
2     A -0.673690
3     B  0.113648
4     A -1.478427
5     A  0.524988
6     B  0.404705
7     A  0.577046
8     A -1.715002
```
Pivot

The Pivot docs.

Partial sums and subtotals

In [151]: df = pd.DataFrame(
   ....:     data={
   ....:         "Province": ["ON", "QC", "BC", "AL", "AL", "MN", "ON"],
   ....:         "City": [
   ....:             "Toronto",
   ....:             "Montreal",
   ....:             "Vancouver",
   ....:             "Calgary",
   ....:             "Edmonton",
   ....:             "Winnipeg",
   ....:             "Windsor",
   ....:         ],
   ....:         "Sales": [13, 6, 16, 8, 4, 3, 1],
   ....:     }
   ....: )

In [152]: table = pd.pivot_table(
   ....:     df,
   ....:     values="Sales",
   ....:     index="Province",
   ....:     columns="City",
   ....:     aggfunc=np.sum,
   ....:     margins=True,
   ....: )

In [153]: table.stack("City")

Out[153]:

<table>
<thead>
<tr>
<th>Province</th>
<th>City</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>All</td>
<td>12.0</td>
</tr>
<tr>
<td></td>
<td>Calgary</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>Edmonton</td>
<td>4.0</td>
</tr>
<tr>
<td>BC</td>
<td>All</td>
<td>16.0</td>
</tr>
<tr>
<td></td>
<td>Vancouver</td>
<td>16.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>All</td>
<td>Montreal</td>
<td>6.0</td>
</tr>
<tr>
<td></td>
<td>Toronto</td>
<td>13.0</td>
</tr>
<tr>
<td></td>
<td>Vancouver</td>
<td>16.0</td>
</tr>
<tr>
<td></td>
<td>Windsor</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>Winnipeg</td>
<td>3.0</td>
</tr>
</tbody>
</table>

[20 rows x 1 columns]

Frequency table like plyr in R

In [154]: grades = [48, 99, 75, 80, 42, 80, 72, 68, 36, 78]

(continues on next page)
In [155]: df = pd.DataFrame(
    .....:     {
    .....:         "ID": ["%d" % r for r in range(10)],
    .....:         "ExamYear": [
    .....:             "2007",
    .....:             "2007",
    .....:             "2007",
    .....:             "2008",
    .....:             "2008",
    .....:             "2008",
    .....:             "2009",
    .....:             "2009",
    .....:             "2009",
    .....:         ],
    .....:         "Class": [
    .....:             "algebra",
    .....:             "stats",
    .....:             "bio",
    .....:             "algebra",
    .....:             "algebra",
    .....:             "stats",
    .....:             "stats",
    .....:             "algebra",
    .....:             "bio",
    .....:             "bio",
    .....:         ],
    .....:         "Participated": [
    .....:             "yes",
    .....:             "yes",
    .....:             "yes",
    .....:             "yes",
    .....:             "no",
    .....:             "yes",
    .....:             "yes",
    .....:             "yes",
    .....:             "yes",
    .....:         ],
    .....:         "Passed": ["yes" if x > 50 else "no" for x in grades],
    .....:         "Employed": [
    .....:             True,
    .....:             True,
    .....:             True,
    .....:             False,
    .....:             False,
    .....:             False,
    .....:             False,
    .....:             True,
    .....:             True,
    .....:         ]
    .....: )
(continued on next page)
.....: False,
.....: ],
.....: "Grade": grades,
.....: }
.....: )
.....:
In [156]: df.groupby("ExamYear").agg(
.....: {  
.....:   "Participated": lambda x: x.value_counts()\["yes"\],
.....:   "Passed": lambda x: sum(x == "yes"),
.....:   "Employed": lambda x: sum(x),
.....:   "Grade": lambda x: sum(x) / len(x),
.....: }
.....: )

Out[156]:

\begin{tabular}{lcccr}
  & Participated & Passed & Employed & Grade \\
ExamYear & & & & \\
2007 & 3 & 2 & 3 & 74.000000 \\
2008 & 3 & 3 & 0 & 68.500000 \\
2009 & 3 & 2 & 2 & 60.666667 \\
\end{tabular}

Plot pandas DataFrame with year over year data

To create year and month cross tabulation:

In [157]: df = pd.DataFrame(
.....: {"value": np.random.randn(36)},
.....: index=pd.date_range("2011-01-01", freq="M", periods=36),
.....: )
.....: 

In [158]: pd.pivot_table(
.....: df, index=df.index.month, columns=df.index.year, values="value", aggfunc=
.....: "sum"
.....: )

Out[158]:

\begin{tabular}{llll}
  & 2011 & 2012 & 2013 \\
1 & -1.039268 & -0.968914 & 2.565646 \\
2 & -0.370647 & -1.294524 & 1.431256 \\
3 & -1.157892 & 0.413738 & 1.340309 \\
4 & -1.344312 & 0.276662 & -1.170299 \\
5 & 0.844885 & -0.472035 & -0.226169 \\
6 & 1.075770 & -0.013960 & 0.410835 \\
7 & -0.109050 & -0.362543 & 0.813850 \\
8 & 1.643563 & -0.006154 & 0.132003 \\
9 & -1.469388 & -0.923061 & -0.827317 \\
10 & 0.357021 & 0.895717 & -0.076467 \\
11 & -0.674600 & 0.805244 & -1.187678 \\
12 & -1.776904 & -1.206412 & 1.130127 \\
\end{tabular}
Apply

Rolling apply to organize - Turning embedded lists into a MultiIndex frame

```
In [159]: df = pd.DataFrame(
    ....:     data={
    ....:         "A": [[2, 4, 8, 16], [100, 200], [10, 20, 30]],
    ....:         "B": [['a', 'b', 'c'], ['jj', 'kk'], ['ccc']],
    ....:     },
    ....:     index=['I', 'II', 'III'],
    ....: )

In [160]: def SeriesFromSubList(aList):
    ....:     return pd.Series(aList)
    ....:

In [161]: df_orgz = pd.concat(
    ....:     {ind: row.apply(SeriesFromSubList) for ind, row in df.iterrows()}
    ....: )

In [162]: df_orgz
Out[162]:
     0  1  2  3
I A 2  4  8 16.0
   B a b c NaN
II A 100 200 NaN NaN
    B jj kk NaN NaN
III A 10 20 30.0 NaN
    B ccc NaN NaN NaN
```

Rolling apply with a DataFrame returning a Series

Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

```
In [163]: df = pd.DataFrame(
    ....:     data=np.random.randn(2000, 2) / 10000,
    ....:     index=pd.date_range("2001-01-01", periods=2000),
    ....:     columns=['A', 'B'],
    ....: )

In [164]: df
Out[164]:
          A         B
2001-01-01 -0.000144 -0.000141
2001-01-02  0.000161  0.000102
2001-01-03  0.000057  0.000088
2001-01-04 -0.000221  0.000097
2001-01-05 -0.000201 -0.000041
...        ...        ...
2006-06-19  0.000040 -0.000235
2006-06-20 -0.000123 -0.000021
```

(continues on next page)
Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)
In [171]: def vwap(bars):
.....:
In [172]: window = 5

In [173]: s = pd.concat(
.....:     [
.....:         pd.Series(vwap(df.iloc[i: i + window]), index=df.index[i + window])
.....:     for i in range(len(df) - window)]
.....: )

In [174]: s.round(2)
Out[174]:
2014-01-06  0.02
2014-01-07  0.11
2014-01-08  0.10
2014-01-09  0.07
2014-01-10  0.29
...  
2014-04-06 -0.63
2014-04-07 -0.02
2014-04-08 -0.03
2014-04-09  0.34
2014-04-10  0.29
Length: 95, dtype: float64
2.27.6 Timeseries

Between times
Using indexer between time
Constructing a datetime range that excludes weekends and includes only certain times
Vectorized Lookup
Aggregation and plotting time series
Turn a matrix with hours in columns and days in rows into a continuous row sequence in the form of a time series. How to rearrange a Python pandas DataFrame?
Dealing with duplicates when reindexing a timeseries to a specified frequency
Calculate the first day of the month for each entry in a DatetimeIndex

```
In [175]: dates = pd.date_range("2000-01-01", periods=5)
In [176]: dates.to_period(freq="M").to_timestamp()
Out[176]:
               '2000-01-01'],
              dtype='datetime64[ns]', freq=None)
```

Resampling

The Resample docs.
Using Grouper instead of TimeGrouper for time grouping of values
Time grouping with some missing values
Valid frequency arguments to Grouper Timeseries
Grouping using a MultiIndex
Using TimeGrouper and another grouping to create subgroups, then apply a custom function GH3791
Resampling with custom periods
Resample intraday frame without adding new days
Resample minute data
Resample with groupby

2.27.7 Merge

The Join docs.
Concatenate two dataframes with overlapping index (emulate R rbind)

```
In [177]: rng = pd.date_range("2000-01-01", periods=6)
In [178]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=["A", "B", "C"])
In [179]: df2 = df1.copy()
```
Depending on df construction, ignore_index may be needed

```
In [180]: df = pd.concat([df1, df2], ignore_index=True)

In [181]: df
Out[181]:
   A    B    C
0 -0.870117 -0.479265 -0.790855
1  0.144817  1.726395 -0.464535
2 -0.821906  1.597605  0.187307
3 -0.128342 -1.511638  0.289858
4  0.399194 -1.430030 -0.639760
5  1.115116 -2.012600  1.810662
6 -0.870117 -0.479265 -0.790855
7  0.144817  1.726395 -0.464535
8 -0.821906  1.597605  0.187307
9 -0.128342 -1.511638  0.289858
10 0.399194 -1.430030 -0.639760
11 1.115116 -2.012600  1.810662
```

Self Join of a DataFrame GH2996

```
In [182]: df = pd.DataFrame(
     .....:     data={
     .....:         "Area": ["A"] * 5 + ["C"] * 2,
     .....:         "Bins": [110] * 2 + [160] * 3 + [40] * 2,
     .....:         "Test_0": [0, 1, 0, 1, 2, 0, 1],
     .....:         "Data": np.random.randn(7),
     .....:     })

In [183]: df
Out[183]:
Area  Bins  Test_0  Data
0   A   110     0  -0.433937
1   A   110     1  -0.160552
2   A   160     0   0.744434
3   A   160     1  1.754213
4   A   160     2  0.000850
5   C    40     0  0.342243
6   C    40     1  1.070599

In [184]: df["Test_1"] = df["Test_0"] - 1

In [185]: pd.merge(
     .....:     df,
     .....:     df,
     .....:     left_on=["Bins", "Area", "Test_0"],
     .....:     right_on=["Bins", "Area", "Test_1"],
     .....:     suffixes=('_L', '_R'),
     .....: )
Out[185]:
```

(continues on next page)
How to set the index and join
KDB like asof join
Join with a criteria based on the values
Using searchsorted to merge based on values inside a range

2.27.8 Plotting

The Plotting docs.
Make Matplotlib look like R
Setting x-axis major and minor labels
Plotting multiple charts in an IPython Jupyter notebook
Creating a multi-line plot
Plotting a heatmap
Annotate a time-series plot
Annotate a time-series plot #2
Generate Embedded plots in excel files using Pandas, Vincent and xlsxwriter
Boxplot for each quartile of a stratifying variable

In [186]: df = pd.DataFrame(
       ...:     {
       ...:         "stratifying_var": np.random.uniform(0, 100, 20),
       ...:         "price": np.random.normal(100, 5, 20),
       ...:     }
       ...: )
       ...

In [187]: df["quartiles"] = pd.qcut(
       ...:     df["stratifying_var"], 4, labels=["0-25%", "25-50%", "50-75%", "75-100%"]
       ...: )
       ...

In [188]: df.boxplot(column="price", by="quartiles")
Out[188]: <AxesSubplot: title={'center': 'price'}, xlabel='quartiles'>
2.27.9 Data in/out

Performance comparison of SQL vs HDF5

CSV

The *CSV* docs

read_csv in action

appending to a csv

Reading a csv chunk-by-chunk

Reading only certain rows of a csv chunk-by-chunk

Reading the first few lines of a frame

Reading a file that is compressed but not by gzip/bz2 (the native compressed formats which read_csv understands). This example shows a WinZipped file, but is a general application of opening the file within a context manager and using that handle to read. See here

Inferring dtypes from a file

Dealing with bad lines GH2886
Write a multi-row index CSV without writing duplicates

**Reading multiple files to create a single DataFrame**

The best way to combine multiple files into a single DataFrame is to read the individual frames one by one, put all of the individual frames into a list, and then combine the frames in the list using `pd.concat()`:

```
In [189]: for i in range(3):
    .....:    data = pd.DataFrame(np.random.randn(10, 4))
    .....:    data.to_csv("file_{}.csv".format(i))
    .....:

In [190]: files = ["file_0.csv", "file_1.csv", "file_2.csv"]

In [191]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

You can use the same approach to read all files matching a pattern. Here is an example using `glob`:

```
In [192]: import glob

In [193]: import os

In [194]: files = glob.glob("file_*.csv")

In [195]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

Finally, this strategy will work with the other `pd.read_*(...)` functions described in the `io docs`.

**Parsing date components in multi-columns**

Parsing date components in multi-columns is faster with a format

```
In [196]: i = pd.date_range("20000101", periods=10000)

In [197]: df = pd.DataFrame({"year": i.year, "month": i.month, "day": i.day})

In [198]: df.head()
```

```
Out[198]:
   year   month  day
0  2000      1   1
1  2000      1   2
2  2000      1   3
3  2000      1   4
4  2000      1   5
```

```
In [199]: %timeit pd.to_datetime(df.year * 10000 + df.month * 100 + df.day, format='%Y%m%d')
```

```
.....: ds = df.apply(lambda x: "%04d%02d%02d" % (x["year"], x["month"], x["day"]), axis=1)
.....: %timeit pd.to_datetime(ds)
```

(continues on next page)
Skip row between header and data

```
In [200]: data = """";;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: ;;;;
.....: date;Param1;Param2;Param4;Param5
.....: m²; °C;m²;m
.....: ;;;;
.....: 01.01.1990 00:00:00;1;1;2;3
.....: 01.01.1990 01:00:00;5;3;4;5
.....: 01.01.1990 02:00:00;9;5;6;7
.....: 01.01.1990 03:00:00;13;7;8;9
.....: 01.01.1990 04:00:00;17;9;10;11
.....: 01.01.1990 05:00:00;21;11;12;13
.....: """
```

Option 1: pass rows explicitly to skip rows

```
In [201]: from io import StringIO

In [202]: pd.read_csv(StringIO(data), sep=";", skiprows=[11, 12], index_col=0, parse_dates=True, header=10)
```

```
Out[202]:
          Param1  Param2  Param4  Param5
date
1990-01-01 00:00:00     1       1       2       3
1990-01-01 01:00:00     5       3       4       5
1990-01-01 02:00:00     9       5       6       7
1990-01-01 03:00:00    13       7       8       9
1990-01-01 04:00:00    17       9      10      11
1990-01-01 05:00:00    21      11      12      13
```
Option 2: read column names and then data

```python
In [203]: pd.read_csv(StringIO(data), sep=";", header=10, nrows=10).columns
Out[203]: Index(['date', 'Param1', 'Param2', 'Param4', 'Param5'], dtype='object')
```

```python
In [204]: columns = pd.read_csv(StringIO(data), sep=";", header=10, nrows=10).columns
In [205]: pd.read_csv(
    ......:    StringIO(data), sep=";", index_col=0, header=12, parse_dates=True,
    ......:    names=columns
    ......: )
Out[205]:
    date       Param1  Param2  Param4  Param5
1990-01-01  00:00:00   1      1      2      3
1990-01-01  01:00:00   5      3      4      5
1990-01-01  02:00:00   9      5      6      7
1990-01-01  03:00:00  13      7      8      9
1990-01-01  04:00:00  17      9     10     11
1990-01-01  05:00:00  21     11     12     13
```

### SQL

The [SQL docs](https://pandas.pydata.org/docs/user_guide/SQL.html)

Reading from databases with SQL

### Excel

The [Excel docs](https://pandas.pydata.org/docs/user_guide/Excel.html)

Reading from a filelike handle

Modifying formatting in XlsxWriter output

Loading only visible sheets GH19842#issuecomment-892150745

### HTML

Reading HTML tables from a server that cannot handle the default request header

### HDFStore

The [HDFStores docs](https://pandas.pydata.org/docs/user_guide/HDFStore.html)

Simple queries with a Timestamp Index

Managing heterogeneous data using a linked multiple table hierarchy GH3032

Merging on-disk tables with millions of rows

Avoiding inconsistencies when writing to a store from multiple processes/threads
De-duplicating a large store by chunks, essentially a recursive reduction operation. Shows a function for taking in data from csv file and creating a store by chunks, with date parsing as well. See here

Creating a store chunk-by-chunk from a csv file

Appending to a store, while creating a unique index

Large Data work flows

Reading in a sequence of files, then providing a global unique index to a store while appending

Groupby on a HDFStore with low group density

Groupby on a HDFStore with high group density

Hierarchical queries on a HDFStore

Counting with a HDFStore

Troubleshoot HDFStore exceptions

Setting min_itemsize with strings

Using ptrepack to create a completely-sorted-index on a store

Storing Attributes to a group node

```
In [206]: df = pd.DataFrame(np.random.randn(8, 3))
In [207]: store = pd.HDFStore("test.h5")
In [208]: store.put("df", df)

# you can store an arbitrary Python object via pickle
In [209]: store.get_storer("df").attrs.my_attribute = {"A": 10}
In [210]: store.get_storer("df").attrs.my_attribute
Out[210]:
{'A': 10}
```

You can create or load a HDFStore in-memory by passing the driver parameter to PyTables. Changes are only written to disk when the HDFStore is closed.

```
In [211]: store = pd.HDFStore("test.h5", "w", driver="H5FD_CORE")
In [212]: df = pd.DataFrame(np.random.randn(8, 3))
In [213]: store["test"] = df

# only after closing the store, data is written to disk:
In [214]: store.close()
```
Binary files

pandas readily accepts NumPy record arrays, if you need to read in a binary file consisting of an array of C structs. For example, given this C program in a file called main.c compiled with gcc main.c -std=gnu99 on a 64-bit machine,

```c
#include <stdio.h>
#include <stdint.h>

typedef struct _Data
{
    int32_t count;
    double avg;
    float scale;
} Data;

int main(int argc, const char *argv[])
{
    size_t n = 10;
    Data d[n];

    for (int i = 0; i < n; ++i)
    {
        d[i].count = i;
        d[i].avg = i + 1.0;
        d[i].scale = (float) i + 2.0f;
    }

    FILE *file = fopen("binary.dat", "wb");
    fwrite(&d, sizeof(Data), n, file);
    fclose(file);

    return 0;
}
```

the following Python code will read the binary file 'binary.dat' into a pandas DataFrame, where each element of the struct corresponds to a column in the frame:

```python
names = "count", "avg", "scale"

# note that the offsets are larger than the size of the type because of
# struct padding
offsets = 0, 8, 16
formats = "i4", "f8", "f4"
dt = np.dtype({"names": names, "offsets": offsets, "formats": formats}, align=True)
df = pd.DataFrame(np.fromfile("binary.dat", dt))
```

Note: The offsets of the structure elements may be different depending on the architecture of the machine on which the file was created. Using a raw binary file format like this for general data storage is not recommended, as it is not cross platform. We recommended either HDF5 or parquet, both of which are supported by pandas’ IO facilities.
2.27.10 Computation

Numerical integration (sample-based) of a time series

Correlation

Often it’s useful to obtain the lower (or upper) triangular form of a correlation matrix calculated from DataFrame.corr(). This can be achieved by passing a boolean mask to where as follows:

```python
In [215]: df = pd.DataFrame(np.random.random(size=(100, 5)))
In [216]: corr_mat = df.corr()
In [217]: mask = np.tril(np.ones_like(corr_mat, dtype=np.bool_), k=-1)
In [218]: corr_mat.where(mask)
```

```
          0   1   2   3   4
0       NaN NaN NaN NaN NaN
1 -0.079861 NaN NaN NaN NaN
2 -0.236573 0.183801 NaN NaN NaN
3 -0.013795 -0.051975 0.037235 NaN NaN
4 -0.031974 0.118342 -0.073499 -0.02063 NaN
```

The method argument within DataFrame.corr can accept a callable in addition to the named correlation types. Here we compute the distance correlation matrix for a DataFrame object.

```python
In [219]: def distcorr(x, y):
   ...:     n = len(x)
   ...:     a = np.zeros(shape=(n, n))
   ...:     b = np.zeros(shape=(n, n))
   ...:     for i in range(n):
   ...:         for j in range(i + 1, n):
   ...:             a[i, j] = abs(x[i] - x[j])
   ...:             b[i, j] = abs(y[i] - y[j])
   ...:     a += a.T
   ...:     b += b.T
   ...:     a_bar = np.vstack([np.nanmean(a, axis=0)] * n)
   ...:     b_bar = np.vstack([np.nanmean(b, axis=0)] * n)
   ...:     A = a - a_bar - a_bar.T + np.full(shape=(n, n), fill_value=a_bar.mean())
   ...:     B = b - b_bar - b_bar.T + np.full(shape=(n, n), fill_value=b_bar.mean())
   ...:     cov_ab = np.sqrt(np.nansum(A * B)) / n
   ...:     std_a = np.sqrt(np.sqrt(np.nansum(A ** 2)) / n)
   ...:     std_b = np.sqrt(np.sqrt(np.nansum(B ** 2)) / n)
   ...:     return cov_ab / std_a / std_b
   ...

In [220]: df = pd.DataFrame(np.random.normal(size=(100, 3)))
In [221]: df.corr(method=distcorr)
```

```
          0   1   2
0 1.000000 0.197613 0.216328
```

(continues on next page)
### 2.27.11 Timedeltas

The *Timedeltas* docs.

Using timedeltas

```
In [222]: import datetime

In [223]: s = pd.Series(pd.date_range("2012-1-1", periods=3, freq="D"))

In [224]: s - s.max()
Out[224]:
0    -2 days
1    -1 days
2     0 days
dtype: timedelta64[ns]

In [225]: s.max() - s
Out[225]:
0     2 days
1     1 days
2     0 days
dtype: timedelta64[ns]

In [226]: s - datetime.datetime(2011, 1, 1, 3, 5)
Out[226]:
0  364 days 20:55:00
1  365 days 20:55:00
2  366 days 20:55:00
dtype: timedelta64[ns]

In [227]: s + datetime.timedelta(minutes=5)
Out[227]:
0  2012-01-01 00:05:00
1  2012-01-02 00:05:00
2  2012-01-03 00:05:00
dtype: datetime64[ns]

In [228]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[228]:
0  -365 days +03:05:00
1  -366 days +03:05:00
2  -367 days +03:05:00
dtype: timedelta64[ns]

In [229]: datetime.timedelta(minutes=5) + s
Out[229]:
0  2012-01-01 00:05:00
```
Adding and subtracting deltas and dates

```
In [230]: deltas = pd.Series([datetime.timedelta(days=i) for i in range(3)])
In [231]: df = pd.DataFrame({"A": s, "B": deltas})
In [232]: df
Out[232]:
   A          B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [233]: df["New Dates"] = df["A"] + df["B"]
In [234]: df["Delta"] = df["A"] - df["New Dates"]
In [235]: df
Out[235]:
   A          B  New Dates  Delta
0 2012-01-01 0 days  2012-01-01 0 days
1 2012-01-02 1 days  2012-01-03 -1 days
2 2012-01-03 2 days  2012-01-05 -2 days
```

Another example

Values can be set to NaT using np.nan, similar to datetime

```
In [237]: y = s - s.shift()
In [238]: y
Out[238]:
0    NaT
1    1 days
2    1 days
dtype: timedelta64[ns]
```

(continues on next page)
2.27.12 Creating example data

To create a dataframe from every combination of some given values, like R’s `expand.grid()` function, we can create a dict where the keys are column names and the values are lists of the data values:

```python
In [241]: def expand_grid(data_dict):
       .....:     rows = itertools.product(*data_dict.values())
       .....:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
       .....:

In [242]: df = expand_grid(
       .....:     {"height": [60, 70], "weight": [100, 140, 180], "sex": ["Male", "Female"]}
       .....: )
       .....:

In [243]: df
Out[243]:
     height  weight  sex
 0    60      100  Male
 1    60      100  Female
 2    60      140   Male
 3    60      140  Female
 4    60      180  Male
 5    60      180  Female
 6    70      100  Male
 7    70      100  Female
 8    70      140   Male
 9    70      140  Female
10   70      180   Male
11   70      180  Female
```
This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in pandas.* namespace are public.

Some subpackages are public which include pandas.errors, pandas.plotting, and pandas.testing. Public functions in pandas.io and pandas.tseries submodules are mentioned in the documentation. pandas.api.types subpackage holds some public functions related to data types in pandas.

**Warning**: The pandas.core, pandas.compat, and pandas.util top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

### 3.1 Input/output

#### 3.1.1 Pickling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_pickle</code></td>
<td>Load pickled pandas object (or any object) from file.</td>
</tr>
<tr>
<td><code>DataFrame.to_pickle</code></td>
<td>Pickle (serialize) object to file.</td>
</tr>
</tbody>
</table>

**pandas.read_pickle**

```python
pandas.read_pickle(filepath_or_buffer[, compression='infer', storage_options=None])
```

Load pickled pandas object (or any object) from file.

**Warning**: Loading pickled data received from untrusted sources can be unsafe. See here.

**Parameters**

- `filepath_or_buffer` [str, path object, or file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a binary `readlines()` function.

  Changed in version 1.0.0: Accept URL. URL is not limited to S3 and GCS.

- `compression` [str or dict, default ‘infer’] For on-the-fly decompression of on-disk data. If ‘infer’ and ‘filepath_or_buffer’ is path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to `None` for no decompression. Can also be a dict with key ‘method’ set to one of {'zip', 'gzip', 'bz2', 'xz', 'zlib', 'zstd'}.
'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary:

```python
compression={'method': 'zstd', 'dict_data': my_compression_dict}
```

Changed in version 1.4.0: Zstandard support.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

Returns

unpickled [same type as object stored in file]

See also:

- **DataFrame.to_pickle** Pickle (serialize) DataFrame object to file.
- **Series.to_pickle** Pickle (serialize) Series object to file.
- **read_hdf** Read HDF5 file into a DataFrame.
- **read_sql** Read SQL query or database table into a DataFrame.
- **read_parquet** Load a parquet object, returning a DataFrame.

Notes

read_pickle is only guaranteed to be backwards compatible to pandas 0.20.3.

Examples

```python
>>> original_df = pd.DataFrame({'foo': range(5), 'bar': range(5, 10)})
>>> original_df
   foo  bar
0   0   5
1   1   6
2   2   7
3   3   8
4   4   9
>>> pd.to_pickle(original_df, './dummy.pkl')

>>> unpickled_df = pd.read_pickle('./dummy.pkl')
>>> unpickled_df
   foo  bar
0   0   5
1   1   6
2   2   7
3   3   8
4   4   9
```
DataFrame.to_pickle(path, compression='infer', protocol=5, storage_options=None)

Pickle (serialize) object to file.

**Parameters**

- **path** [str] File path where the pickled object will be stored.
- **compression** [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘path’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of ['zip', 'gzip', 'bz2', 'zstd'] and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.
- **protocol** [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1] paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.
- **storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

See also:

- **read_pickle** Load pickled pandas object (or any object) from file.
- **DataFrame.to_hdf** Write DataFrame to an HDF5 file.
- **DataFrame.to_sql** Write DataFrame to a SQL database.
- **DataFrame.to_parquet** Write a DataFrame to the binary parquet format.

**Examples**

```python
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0   0   5
1   1   6
2   2   7
3   3   8
4   4   9
>>> original_df.to_pickle("./dummy.pkl")
```

```python
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0   0   5
```

(continues on next page)
3.1.2 Flat file

<table>
<thead>
<tr>
<th>read_table(filepath_or_buffer[, sep, ...])</th>
<th>Read general delimited file into DataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_csv(filepath_or_buffer[, sep, ...])</td>
<td>Read a comma-separated values (csv) file into DataFrame.</td>
</tr>
<tr>
<td>DataFrame.to_csv([path_or_buf, sep, na_rep, ...])</td>
<td>Write object to a comma-separated values (csv) file.</td>
</tr>
<tr>
<td>read_fwf(filepath_or_buffer[, colspecs, ...])</td>
<td>Read a table of fixed-width formatted lines into DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.read_table**

```python
pandas.read_table(filepath_or_buffer, sep=None, delimiter=None, header='infer',
names=None, index_col=None, usecols=None, squeeze=False,
prefix=None, mangle_dupe_cols=True, dtype=None, engine=None,
converters=None, true_values=None, false_values=None,
skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None,
na_values=None, keep_default_na=True, na_filter=True, verbose=False,
skip_blank_lines=True, parse_dates=False, infer_datetime_format=False,
keep_date_col=False, date_parser=None, dayfirst=False,
cache_dates=True, iterator=False, chunksize=None,
compression=None, thousands=None,
decimal=',', lineterminator=None, quotechar=None, quoting=0,
doublequote=True, escapechar=None, comment=None, encoding=None,
encoding_errors='strict', dialect=None, error_bad_lines=None,
warn_bad_lines=None, on_bad_lines=None, delim_whitespace=False,
low_memory=True, memory_map=False, float_precision=None,
storage_options=None)
```

Read general delimited file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

**Parameters**

- **filepath_or_buffer** [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

  If you want to pass in a path object, pandas accepts any os.PathLike.

  By file-like object, we refer to objects with a read() method, such as a file handle (e.g. via builtin open function) or StringIO.

- **sep** [str, default ‘\t’ (tab-stop)] Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s builtin sniffer tool, csv.Sniffer. In addition, separators longer than 1 character and different from ‘\s+’ will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: ‘\r\t’.
delimiter [str, default None] Alias for sep.

header [int, list of int, None, default 'infer'] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None. Explicitly pass header=0 to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if skip_blank_lines=True, so header=0 denotes the first line of data rather than the first line of the file.

names [array-like, optional] List of column names to use. If the file contains a header row, then you should explicitly pass header=0 to override the column names. Duplicates in this list are not allowed.

index_col [int, str, sequence of int / str, or False, optional, default None] Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: index_col=False can be used to force pandas to not use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

usecols [list-like or callable, optional] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If names are given, the document header row(s) are not taken into account. For example, a valid list-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz']. Element order is ignored, so usecols=[0, 1] is the same as [1, 0]. To instantiate a DataFrame from data with element order preserved use pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']] for columns in ['foo', 'bar'] order or pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be lambda x: x.upper() in ['AAA', 'BBB', 'DDD']. Using this parameter results in much faster parsing time and lower memory usage.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

Deprecated since version 1.4.0: Append .squeeze("columns") to the call to read_table to squeeze the data.

prefix [str, optional] Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, …

Deprecated since version 1.4.0: Use a list comprehension on the DataFrame’s columns after calling read_csv.

mangle_dupe_cols [bool, default True] Duplicate columns will be specified as ‘X’, ‘X.1’, … ‘X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

dtype [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use str or object together with suitable na_values settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [{'c', 'python', 'pyarrow'}, optional] Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is
converters [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, optional] Values to consider as True.

false_values [list, optional] Values to consider as False.

skipinitialspace [bool, default False] Skip spaces after delimiter.

skiprows [list-like, int or callable, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

    If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be lambda x: x in [0, 2].

skipfooter [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

nrows [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

na_values [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: "", "N/A", "N/A N/A", "NA", "-1.#IND", "-1.#QNaN", "-NaN", "-nan", "1.#IND", "1.#QNaN", "<NA>", 'N/A', 'NA', 'NULL', ' NaN', 'nan', 'null'.

keep_default_na [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether na_values is passed in, the behavior is as follows:

    • If keep_default_na is True, and na_values are specified, na_values is appended to the default NaN values used for parsing.

    • If keep_default_na is True, and na_values are not specified, only the default NaN values are used for parsing.

    • If keep_default_na is False, and na_values are specified, only the NaN values specified na_values are used for parsing.

    • If keep_default_na is False, and na_values are not specified, no strings will be parsed as NaN.

    Note that if na_filter is passed in as False, the keep_default_na and na_values parameters will be ignored.

na_filter [bool, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

parse_dates [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

    • boolean. If True -> try parsing the index.

    • list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
• list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
• dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index cannot be represented as an array of datetimes, say because of an un-parseable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv. To parse an index or column with a mixture of timezones, specify date_parser to be a partially-applied pandas.to_datetime() with utc=True. See Parsing a CSV with mixed timezones for more.

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format [bool, default False] If True and parse_dates is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_col [bool, default False] If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parser [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion. Pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirst [bool, default False] DD/MM format dates, international and European format.

cache_dates [bool, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

iterator [bool, default False] Return TextFileReader object for iteration or getting chunks with get_chunk().

Changed in version 1.2: TextFileReader is a context manager.

chunksize [int, optional] Return TextFileReader object for iteration. See the IO Tools docs for more information on iterator and chunksize.

Changed in version 1.2: TextFileReader is a context manager.

compression [str or dict, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer' and '%%s' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', or '.zst' (otherwise no compression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key 'method' set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: compression={'method': 'zstd', 'dict_data': my_compression_dict}.

Changed in version 1.4.0: Zstandard support.

thousands [str, optional] Thousands separator.

decimal [str, default ‘.’] Character to recognize as decimal point (e.g. use ‘,’ for European data).
lineterminator  [str (length 1), optional] Character to break file into lines. Only valid with C parser.

quotechar  [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting  [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequote  [bool, default True] When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements INSIDE a field as a single quotechar element.

escapechar  [str (length 1), optional] One-character string used to escape other characters.

comment  [str, optional] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as skip_blank_lines=True), fully commented lines are ignored by the parameter header but not by skiprows. For example, if comment='\#', parsing "\#empty\n\na,b,c\n\n1,2,3" with header=0 will result in ‘a,b,c’ being treated as the header.

encoding  [str, optional] Encoding to use for UTF when reading/writing (ex. ‘utf-8’). List of Python standard encodings .

Changed in version 1.2: When encoding is None, errors="replace" is passed to open(). Otherwise, errors="strict" is passed to open(). This behavior was previously only the case for engine="python".

Changed in version 1.3.0: encoding_errors is a new argument. encoding has no longer an influence on how encoding errors are handled.

encoding_errors  [str, optional, default “strict”] How encoding errors are treated. List of possible values .

New in version 1.3.0.

dialect  [str or csv.Dialect, optional] If provided, this parameter will override values (default or not) for the following parameters: delimiter, doublequote, escapechar, skipinitialspace, quotechar, and quoting. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

error_bad_lines  [bool, optional, default None] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these “bad lines” will be dropped from the DataFrame that is returned.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

warn_bad_lines  [bool, optional, default None] If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

on_bad_lines  [{'error', 'warn', 'skip'} or callable, default ‘error’] Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are:

• ‘error’, raise an Exception when a bad line is encountered.
• ‘warn’, raise a warning when a bad line is encountered and skip that line.
• ‘skip’, skip bad lines without raising or warning when they are encountered.

New in version 1.3.0:
• callable, function with signature (bad_line: list[str]) \rightarrow list[str] | None that will process a single bad line. bad_line is a list of strings split by the sep. If the function returns None, the bad line will be ignored. If the function returns a new list of strings with more elements than expected, a ParserWarning will be emitted while dropping extra elements. Only supported when engine="python"

New in version 1.4.0.

delim_whitespace [bool, default False] Specifies whether or not whitespace (e.g. ' ' or '\') will be used as the sep. Equivalent to setting sep='\s+'. If this option is set to True, nothing should be passed in for the delimiter parameter.

low_memory [bool, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser).

memory_map [bool, default False] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

float_precision [str, optional] Specifies which converter the C engine should use for floating-point values. The options are None or 'high' for the ordinary converter, 'legacy' for the original lower precision pandas converter, and 'round_trip' for the round-trip converter.

Changed in version 1.2.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

DataFrame.to_csv Write DataFrame to a comma-separated values (csv) file.
read_csv Read a comma-separated values (csv) file into DataFrame.
read_fwf Read a table of fixed-width formatted lines into DataFrame.
Examples

```python
>>> pd.read_table('data.csv')
```

pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.read_csv

```python
pandas.read_csv(filepath_or_buffer, sep=None, delimiter=None, names=None, index_col=None, usecols=None, squeeze=None, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=None, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression=None, thousands=None, decimal='', lineterminator=None, quotechar=None, quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, error_bad_lines=None, warn_bad_lines=None, on_bad_lines=None, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None)
```

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

**Parameters**

- `filepath_or_buffer` [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

  If you want to pass in a path object, pandas accepts any `os.PathLike`.

  By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via built-in open function) or `StringIO`.

- `sep` [str, default `','`] Delimiter to use. If `sep` is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s built-in sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `\s+` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\t'`.

- `delimiter` [str, default `None`] Alias for `sep`.

- `header` [int, list of int, None, default `'infer'`] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

- `names` [array-like, optional] List of column names to use. If the file contains a header row, then you should explicitly pass `header=0` to override the column names. Duplicates in this list are not allowed.
**index_col** [int, str, sequence of int / str, or False, optional, default None] Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: **index_col=False** can be used to force pandas to not use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

**usecols** [list-like or callable, optional] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If names are given, the document header row(s) are not taken into account. For example, a valid list-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz']. Element order is ignored, so usecols=[0, 1] is the same as [1, 0]. To instantiate a DataFrame from data with element order preserved use pd.read_csv(data, usecols=['foo', 'bar'])[[‘foo’, ‘bar’]] for columns in ['foo', 'bar'] order or pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be lambda x: x.upper() in ['AAA', 'BBB', 'DDD']. Using this parameter results in much faster parsing time and lower memory usage.

**squeeze** [bool, default False] If the parsed data only contains one column then return a Series.

Deprecated since version 1.4.0: Append .squeeze(“columns”) to the call to read_csv to squeeze the data.

**prefix** [str, optional] Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, …

Deprecated since version 1.4.0: Use a list comprehension on the DataFrame’s columns after calling read_csv.

**mangle_dupe_cols** [bool, default True] Duplicate columns will be specified as ‘X’, ‘X.1’, … ‘X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**dtype** [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {‘a’: np.float64, ‘b’: np.int32, ‘c’: ‘Int64’} Use str or object together with suitable na_values settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**engine** [{'c', 'python', 'pyarrow'}, optional] Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

New in version 1.4.0: The “pyarrow” engine was added as an experimental engine, and some features are unsupported, or may not work correctly, with this engine.

**converters** [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

**true_values** [list, optional] Values to consider as True.

**false_values** [list, optional] Values to consider as False.

**skipinitialspace** [bool, default False] Skip spaces after delimiter.

**skiprows** [list-like, int or callable, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.
If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

**skipfooter** [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

**nrows** [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

**na_values** [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: ‘’, ‘#N/A’, ‘#N/A N/A’, ‘#NA’, ‘-1.#IND’, ‘-1.#QNAN’, ‘-NaN’, ‘-nan’, ‘1.#IND’, ‘1.#QNAN’, ‘<NA>’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

**keep_default_na** [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether na_values is passed in, the behavior is as follows:

- If `keep_default_na` is True, and na_values are specified, na_values is appended to the default NaN values used for parsing.
- If `keep_default_na` is True, and na_values are not specified, only the default NaN values are used for parsing.
- If `keep_default_na` is False, and na_values are specified, only the NaN values specified na_values are used for parsing.
- If `keep_default_na` is False, and na_values are not specified, no strings will be parsed as NaN.

Note that if na_filter is passed in as False, the keep_default_na and na_values parameters will be ignored.

**na_filter** [bool, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

**verbose** [bool, default False] Indicate number of NA values placed in non-numeric columns.

**skip_blank_lines** [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

**parse_dates** [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1], [3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. `{‘foo’ : [1, 3]}` -> parse columns 1, 3 as date and call result ‘foo’

If a column or index cannot be represented as an array of datetimes, say because of an unparsable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv. To parse an index or column with a mixture of timezones, specify date_parser to be a partially-applied pandas.to_datetime() with utc=True. See Parsing a CSV with mixed timezones for more.

Note: A fast-path exists for iso8601-formatted dates.

**infer_datetime_format** [bool, default False] If True and parse_dates is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred,
switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

**keep_date_col** [bool, default False] If True and `parse_dates` specifies combining multiple columns then keep the original columns.

**date_parser** [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call `date_parser` in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

**dayfirst** [bool, default False] DD/MM format dates, international and European format.

**cache_dates** [bool, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

**iterator** [bool, default False] Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

Changed in version 1.2: `TextFileReader` is a context manager.

**chunksize** [int, optional] Return `TextFileReader` object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

Changed in version 1.2: `TextFileReader` is a context manager.

**compression** [str or dict, default ‘infer’] For on-the-fly decompression of on-disk data. If ‘infer’ and ‘%s’ is path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key ‘method’ set to one of [‘zip’, ‘gzip’, ‘bz2’, ‘zstd’] and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: `compression={'method': 'zstd', 'dict_data': my_compression_dict}`.

Changed in version 1.4.0: Zstandard support.

**thousands** [str, optional] Thousands separator.

**decimal** [str, default ‘.’] Character to recognize as decimal point (e.g. use ‘,’ for European data).

**linterterminator** [str (length 1), optional] Character to break file into lines. Only valid with C parser.

**quotechar** [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**quoting** [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

**doublequote** [bool, default True] When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements INSIDE a field as a single quotechar element.

**escapechar** [str (length 1), optional] One-character string used to escape other characters.
**comment** [str, optional] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in ‘a,b,c’ being treated as the header.

**encoding** [str, optional] Encoding to use for UTF when reading/writing (ex. ‘utf-8’). List of Python standard encodings.

    Changed in version 1.2: When `encoding` is `None`, `errors="replace"` is passed to `open()`. Otherwise, `errors="strict"` is passed to `open()`. This behavior was previously only the case for `engine="python"`.

    Changed in version 1.3.0: `encoding_errors` is a new argument. `encoding` has no longer an influence on how encoding errors are handled.

**encoding_errors** [str, optional, default “strict”] How encoding errors are treated. List of possible values.

    New in version 1.3.0.

**dialect** [str or csv.Dialect, optional] If provided, this parameter will override values (default or not) for the following parameters: `delimiter`, `doublequote`, `escapechar`, `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

**error_bad_lines** [bool, optional, default `None`] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these “bad lines” will be dropped from the DataFrame that is returned.

    Deprecated since version 1.3.0: The `on_bad_lines` parameter should be used instead to specify behavior upon encountering a bad line instead.

**warn_bad_lines** [bool, optional, default `None`] If `error_bad_lines` is False, and `warn_bad_lines` is True, a warning for each “bad line” will be output.

    Deprecated since version 1.3.0: The `on_bad_lines` parameter should be used instead to specify behavior upon encountering a bad line instead.

**on_bad_lines** [{‘error’, ‘warn’, ‘skip’} or callable, default ‘error’] Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are:

  • ‘error’, raise an Exception when a bad line is encountered.
  • ‘warn’, raise a warning when a bad line is encountered and skip that line.
  • ‘skip’, skip bad lines without raising or warning when they are encountered.

    New in version 1.3.0:

  • callable, function with signature `(bad_line: list[str]) -> list[str] | None` that will process a single bad line. `bad_line` is a list of strings split by the `sep`. If the function returns `None`, the bad line will be ignored. If the function returns a new list of strings with more elements than expected, a `ParserWarning` will be emitted while dropping extra elements. Only supported when `engine="python"`

    New in version 1.4.0.

**delim_whitespace** [bool, default False] Specifies whether or not whitespace (e.g. ‘ ‘ or ‘ ’) will be used as the `sep`. Equivalent to setting `sep=\s+`. If this option is set to True, nothing should be passed in for the `delimiter` parameter.
**low_memory** [bool, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the *dtype* parameter. Note that the entire file is read into a single DataFrame regardless, use the *chunksize* or *iterator* parameter to return the data in chunks. (Only valid with C parser).

**memory_map** [bool, default False] If a filepath is provided for *filepath_or_buffer*, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

**float_precision** [str, optional] Specifies which converter the C engine should use for floating-point values. The options are *None* or ‘high’ for the ordinary converter, ‘legacy’ for the original lower precision pandas converter, and ‘round_trip’ for the round-trip converter.

Changed in version 1.2.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to *urllib* as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to *fsspec*. Please see *fsspec* and *urllib* for more details.

New in version 1.2.

**Returns**

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

*DataFrame.to_csv* Write DataFrame to a comma-separated values (csv) file.

*read_csv* Read a comma-separated values (csv) file into DataFrame.

*read_fwf* Read a table of fixed-width formatted lines into DataFrame.

**Examples**

```python
>>> pd.read_csv('data.csv')
```

**pandas.DataFrame.to_csv**

DataFrame.to_csv(*path_or_buf=None*, *sep='*, *na_rep='*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*), *encoding=None*, *compression=None*, *quoting=None*, *quotechar='", *line_terminator=None*, *chunksize=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*, *errors='strict'*, *storage_options=None*)

Write object to a comma-separated values (csv) file.

**Parameters**

*path_or_buf* [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with *newlines='*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a ‘b’.

Changed in version 1.2.0: Support for binary file objects was introduced.
sep [str, default ‘,’] String of length 1. Field delimiter for the output file.

na_rep [str, default ‘’] Missing data representation.

float_format [str, default None] Format string for floating point numbers.

columns [sequence, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

index_label [str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and header and index are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode [str] Python write mode, default ‘w’.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to ‘utf-8’. encoding is not supported if path_or_buf is a non-binary file object.

compression [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘%s’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

Changed in version 1.0.0: May now be a dict with key ‘method’ as compression mode and other entries as additional compression options if compression mode is ‘zip’.

Changed in version 1.1.0: Passing compression options as keys in dict is supported for compression modes ‘gzip’, ‘bz2’, ‘zstd’, and ‘zip’.

Changed in version 1.2.0: Compression is supported for binary file objects.

Changed in version 1.2.0: Previous versions forwarded dict entries for ‘gzip’ to gzip.open instead of gzip.GzipFile which prevented setting mtime.

quoting [optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a float_format then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar [str, default ‘’] String of length 1. Character used to quote fields.

line_terminator [str, optional] The newline character or character sequence to use in the output file. Defaults to os.linesep, which depends on the OS in which this method is called (‘\n’ for linux, ‘\r\n’ for Windows, i.e.).

chunksize [int or None] Rows to write at a time.

date_format [str, default None] Format string for datetime objects.

doublequote [bool, default True] Control quoting of quotechar inside a field.

escapechar [str, default None] String of length 1. Character used to escape sep and quotechar when appropriate.

decimal [str, default ‘.’] Character recognized as decimal separator. E.g. use ‘,’ for European data.
errors [str, default ‘strict’] Specifies how encoding and decoding errors are to be handled. See the errors argument for open() for a full list of options.

New in version 1.1.0.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

Returns

None or str If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

read_csv Load a CSV file into a DataFrame.
to_excel Write DataFrame to an Excel file.

Examples

```python
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                     'mask': ['red', 'purple'],
...                     'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon
Raphael,red,sai
Donatello,purple,bo staff

Create 'out.zip' containing 'out.csv'

``` python
>>> compression_opts = dict(method='zip',
...                           archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```python
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)

>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

**pandas.read_fwf**

```python
pandas.read_fwf(filepath_or_buffer, colspecs='infer', widths=None, infer_nrows=100, **kwds)
```

Read a table of fixed-width formatted lines into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

**Parameters**

- `filepath_or_buffer` [str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a text `read()` function. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.csv`.

- `colspecs` [list of tuple (int, int) or 'infer', optional] A list of tuples giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value ‘infer’ can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data which are not being skipped via `skiprows` (default='infer').

- `widths` [list of int, optional] A list of field widths which can be used instead of `colspecs` if the intervals are contiguous.

- `infer_nrows` [int, default 100] The number of rows to consider when letting the parser determine the `colspecs`.

- `**kwds` [optional] Optional keyword arguments can be passed to `TextFileReader`.

**Returns**

- DataFrame or TextFileReader A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

**See also:**

- `DataFrame.to_csv` Write DataFrame to a comma-separated values (csv) file.
- `read_csv` Read a comma-separated values (csv) file into DataFrame.

**Examples**

```python
>>> pd.read_fwf('data.csv')
```

### 3.1.3 Clipboard

<table>
<thead>
<tr>
<th>read_clipboard([sep])</th>
<th>Read text from clipboard and pass to <code>read_csv</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.to_clipboard([excel, sep])</td>
<td>Copy object to the system clipboard.</td>
</tr>
</tbody>
</table>
**pandas.read_clipboard**

```python
def pandas.read_clipboard(sep='\s+', **kwargs):
    Read text from clipboard and pass to read_csv.
```

**Parameters**

- `sep` [str, default 's+'] A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.
- `**kwargs` See read_csv for the full argument list.

**Returns**

- `DataFrame` A parsed DataFrame object.

**pandas.DataFrame.to_clipboard**

```python
def pandas.DataFrame.to_clipboard(excel=True, sep=None, **kwargs):
    Copy object to the system clipboard.
```

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

**Parameters**

- `excel` [bool, default True] Produce output in a csv format for easy pasting into excel.
  - True, use the provided separator for csv pasting.
  - False, write a string representation of the object to the clipboard.
- `sep` [str, default '\t'] Field delimiter.
- `**kwargs` These parameters will be passed to DataFrame.to_csv.

**See also:**

- `DataFrame.to_csv` Write a DataFrame to a comma-separated values (csv) file.
- `read_clipboard` Read text from clipboard and pass to read_csv.

**Notes**

Requirements for your platform.

- Linux : xclip, or xsel (with PyQt4 modules)
- Windows : none
- macOS : none
Examples

Copy the contents of a DataFrame to the clipboard.

```python
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
```

```python
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword `index` and setting it to false.

```python
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

### 3.1.4 Excel

<table>
<thead>
<tr>
<th>pandas.read_excel</th>
<th>Read an Excel file into a pandas DataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td>io</td>
<td>Read an Excel file into a pandas DataFrame.</td>
</tr>
<tr>
<td>sheet_name</td>
<td>sheet name of the sheet to read.</td>
</tr>
<tr>
<td>header</td>
<td>Whether to read the first row as column names.</td>
</tr>
<tr>
<td>names</td>
<td>List of column names.</td>
</tr>
<tr>
<td>index_col</td>
<td>Column index to read.</td>
</tr>
<tr>
<td>usecols</td>
<td>List of column names to read.</td>
</tr>
<tr>
<td>squeeze</td>
<td>Whether to squeeze the data.</td>
</tr>
<tr>
<td>dtype</td>
<td>Data type to use.</td>
</tr>
<tr>
<td>engine</td>
<td>Engine to use for reading.</td>
</tr>
<tr>
<td>converters</td>
<td>Dictionary of converters.</td>
</tr>
<tr>
<td>true_values</td>
<td>True values to use.</td>
</tr>
<tr>
<td>false_values</td>
<td>False values to use.</td>
</tr>
<tr>
<td>skiprows</td>
<td>Rows to skip.</td>
</tr>
<tr>
<td>nrows</td>
<td>Number of rows to read.</td>
</tr>
<tr>
<td>na_values</td>
<td>Values to use for NA.</td>
</tr>
<tr>
<td>keep_default_na</td>
<td>Whether to keep default NaN.</td>
</tr>
<tr>
<td>na_filter</td>
<td>Whether to filter NA.</td>
</tr>
<tr>
<td>verbose</td>
<td>Whether to show verbose output.</td>
</tr>
<tr>
<td>parse_dates</td>
<td>Whether to parse dates.</td>
</tr>
<tr>
<td>date_parser</td>
<td>Date parser function.</td>
</tr>
<tr>
<td>thousands</td>
<td>Thousands separator.</td>
</tr>
<tr>
<td>decimal</td>
<td>Decimal separator.</td>
</tr>
<tr>
<td>comment</td>
<td>Comment separator.</td>
</tr>
<tr>
<td>skipfooter</td>
<td>Number of footer rows to skip.</td>
</tr>
<tr>
<td>convert_float</td>
<td>Whether to convert floats.</td>
</tr>
<tr>
<td>mangle_dupe_cols</td>
<td>Whether to mangle duplicate columns.</td>
</tr>
<tr>
<td>storage_options</td>
<td>Storage options.</td>
</tr>
</tbody>
</table>

**pandas.read_excel**

Read an Excel file into a pandas DataFrame.

Supports `.xls`, `.xlsx`, `.xlsm`, `.xlsb`, `.odf`, `.ods`, and `.odt` file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.

**Parameters**

- **io** [str, bytes, ExcelFile, xlr.Book, path object, or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.xlsx`.

  If you want to pass in a path object, pandas accepts any `os.PathLike`.

  By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via `builtin.open function`) or `StringIO`.

- **sheet_name** [str, int, list, or None, default 0] Strings are used for sheet names. Integers are used in zero-indexed sheet positions (chart sheets do not count as a sheet position). Lists of strings/integers are used to request multiple sheets. Specify None to get all worksheets.
Available cases:

- Defaults to 0: 1st sheet as a DataFrame
- 1: 2nd sheet as a DataFrame
- "Sheet1": Load sheet with name “Sheet1”
- [0, 1, "Sheet5"]: Load first, second and sheet named “Sheet5” as a dict of DataFrame
- None: All worksheets.

header [int, list of int, default 0] Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a MultiIndex. Use None if there is no header.

names [array-like, default None] List of column names to use. If file contains no header row, then you should explicitly pass header=None.

index_col [int, list of int, default None] Column (0-indexed) to use as the row labels of the DataFrame. Pass None if there is no such column. If a list is passed, those columns will be combined into a MultiIndex. If a subset of data is selected with usecols, index_col is based on the subset.

usecols [int, str, list-like, or callable default None]

- If None, then parse all columns.
- If str, then indicates comma separated list of Excel column letters and column ranges (e.g. “A:E” or “A,C,E:F”). Ranges are inclusive of both sides.
- If list of int, then indicates list of column numbers to be parsed.
- If list of string, then indicates list of column names to be parsed.
- If callable, then evaluate each column name against it and parse the column if the callable returns True.

Returns a subset of the columns according to behavior above.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

Depreciated since version 1.4.0: Append .squeeze("columns") to the call to read_excel to squeeze the data.

dtype [Type name or dict of column -> type, default None] Data type for data or columns. E.g. 

| 'a': np.float64, 'b': np.int32 | Use object to preserve data as stored in Excel and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [str, default None] If io is not a buffer or path, this must be set to identify io. Supported engines: “xlrd”, “openpyxl”, “odf”, “pyxlsb”. Engine compatibility :

- “xlrd” supports old-style Excel files (.xls).
- “openpyxl” supports newer Excel file formats.
- “odf” supports OpenDocument file formats (.odf, .ods, .odt).
- “pyxlsb” supports Binary Excel files.

Changed in version 1.2.0: The engine xlrd now only supports old-style .xls files. When engine=None, the following logic will be used to determine the engine:

- If path_or_buffer is an OpenDocument format (.odf, .ods, .odt), then odf will be used.
- Otherwise if path_or_buffer is an xls format, xlrd will be used.
- Otherwise if `path_or_buffer` is in xlsb format, `pyxlsb` will be used.
  New in version 1.3.0.
- Otherwise `openpyxl` will be used.
  Changed in version 1.3.0.

**converters** [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the Excel cell content, and return the transformed content.

**true_values** [list, default None] Values to consider as True.

**false_values** [list, default None] Values to consider as False.

**skiprows** [list-like, int, or callable, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file. If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

**nrows** [int, default None] Number of rows to parse.

**na_values** [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: ‘’, ‘#N/A’, ‘#N/A N/A’, ‘#NA’, ‘-1.#IND’, ‘-1.#QNAN’, ‘-NaN’, ‘-nan’, ‘1.#IND’, ‘1.#QNAN’, ‘<NA>’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

**keep_default_na** [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether `na_values` is passed in, the behavior is as follows:

- If `keep_default_na` is True, and `na_values` are specified, `na_values` is appended to the default NaN values used for parsing.
- If `keep_default_na` is True, and `na_values` are not specified, only the default NaN values are used for parsing.
- If `keep_default_na` is False, and `na_values` are specified, only the NaN values specified `na_values` are used for parsing.
- If `keep_default_na` is False, and `na_values` are not specified, no strings will be parsed as NaN.

Note that if `na_filter` is passed in as False, the `keep_default_na` and `na_values` parameters will be ignored.

**na_filter** [bool, default True] Detect missing value markers (empty strings and the value of `na_values`). In data without any NAs, passing `na_filter=False` can improve the performance of reading a large file.

**verbose** [bool, default False] Indicate number of NA values placed in non-numeric columns.

**parse_dates** [bool, list-like, or dict, default False] The behavior is as follows:

- bool. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {‘foo’ : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’

If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. If you don’t want to parse some cells as date just change
their type in Excel to “Text”. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_excel`.

Note: A fast-path exists for iso8601-formatted dates.

**date_parser** [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call `date_parser` in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

**thousands** [str, default None] Thousands separator for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.

**decimal** [str, default '.'] Character to recognize as decimal point for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.(e.g. use ‘,’ for European data).

New in version 1.4.0.

**comment** [str, default None] Comments out remainder of line. Pass a character or characters to this argument to indicate comments in the input file. Any data between the comment string and the end of the current line is ignored.

**skipfooter** [int, default 0] Rows at the end to skip (0-indexed).

**convert_float** [bool, default True] Convert integral floats to int (i.e., 1.0 -> 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally.

Deprecated since version 1.3.0: `convert_float` will be removed in a future version

**mangle_dupe_cols** [bool, default True] Duplicate columns will be specified as ‘X’, ‘X.1’, … ‘X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc., if using a URL that will be parsed by `fsspec`, e.g., starting “s3://”, “gcs://”. An error will be raised if providing this argument with a local path or a file-like buffer. See the `fsspec` and backend storage implementation docs for the set of allowed keys and values.

New in version 1.2.0.

**Returns**

**DataFrame or dict of DataFrames** DataFrame from the passed in Excel file. See notes in `sheet_name` argument for more information on when a dict of DataFrames is returned.

**See also:**

`DataFrame.to_excel` Write DataFrame to an Excel file.

`DataFrame.to_csv` Write DataFrame to a comma-separated values (csv) file.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

`read_fwf` Read a table of fixed-width formatted lines into DataFrame.
Examples

The file can be read using the file name as string or an open file object:

```python
>>> pd.read_excel('tmp.xlsx', index_col=0)
Name  Value
0    string1  1
1    string2  2
2   #Comment  3
```

```python
>>> pd.read_excel(open('tmp.xlsx', 'rb'), ... sheet_name='Sheet3')
Index: Unnamed: 0   Name  Value
     0      0  string1  1
     1      1  string2  2
     2      2   #Comment  3
```

Index and header can be specified via the `index_col` and `header` arguments

```python
>>> pd.read_excel('tmp.xlsx', index_col=None, header=None)
0  1  2
0  NaN Name  Value
1  0.0  string1  1
2  1.0  string2  2
3  2.0  #Comment  3
```

Column types are inferred but can be explicitly specified

```python
>>> pd.read_excel('tmp.xlsx', index_col=0, ... dtype={'Name': str, 'Value': float})
Name  Value
0    string1  1.0
1    string2  2.0
2   #Comment  3.0
```

True, False, and NA values, and thousands separators have defaults, but can be explicitly specified, too. Supply the values you would like as strings or lists of strings!

```python
>>> pd.read_excel('tmp.xlsx', index_col=0, ... na_values=['string1', 'string2'])
Name  Value
0   NaN  1
1   NaN  2
2   #Comment  3
```

Comment lines in the excel input file can be skipped using the `comment` kwarg

```python
>>> pd.read_excel('tmp.xlsx', index_col=0, comment='#')
Name  Value
0    string1  1.0
1    string2  2.0
2    None    NaN
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.DataFrame.to_excel

`DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep='\', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options=None)`

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an ExcelWriter object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique sheet_name. With all data written to the file it is necessary to save the changes. Note that creating an ExcelWriter object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

- `excel_writer` [path-like, file-like, or ExcelWriter object] File path or existing ExcelWriter.
- `sheet_name` [str, default 'Sheet1'] Name of sheet which will contain DataFrame.
- `na_rep` [str, default '\'] Missing data representation.
- `float_format` [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.
- `columns` [sequence or list of str, optional] Columns to write.
- `header` [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.
- `index` [bool, default True] Write row names (index).
- `index_label` [str or sequence, optional] Column label for index column(s) if desired. If not specified, and header and index are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- `startrow` [int, default 0] Upper left cell row to dump data frame.
- `startcol` [int, default 0] Upper left cell column to dump data frame.

Deprecated since version 1.2.0: As the xlwt package is no longer maintained, the xlwt engine will be removed in a future version of pandas.

- `merge_cells` [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.
- `encoding` [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.
- `inf_rep` [str, default ‘inf’] Representation for infinity (there is no native representation for infinity in Excel).
- `verbose` [bool, default True] Display more information in the error logs.
- `freeze_panes` [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.
- `storage_options` [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are
forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

See also:

to_csv Write DataFrame to a comma-separated values (csv) file.

ExcelWriter Class for writing DataFrame objects into excel sheets.

read_excel Read an Excel file into a pandas DataFrame.

read_csv Read a comma-separated values (csv) file into DataFrame.

Notes

For compatibility with to_csv(), to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```python
>>> df1 = pd.DataFrame([[a, b], [c, d]],
                      index=['row 1', 'row 2'],
                      columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```python
>>> df1.to_excel("output.xlsx",
               sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```python
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```python
>>> with pd.ExcelWriter('output.xlsx',
                      mode='a') as writer:
...     df.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the engine keyword (the default engine is automatically chosen depending on the file extension):

```python
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```
ExcelFile.parse

ExcelFile.parse(sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, convert_float=None, mangle_dupe_cols=True, **kwds)

Parse specified sheet(s) into a DataFrame.

Equivalent to read_excel(ExcelFile, ...). See the read_excel docstring for more info on accepted parameters.

Returns

DataFrame or dict of DataFrames  DataFrame from the passed in Excel file.

Styler.to_excel(excel_writer[, sheet_name, ...])  Write Styler to an Excel sheet.

Styler.to_excel(excel_writer, sheet_name='Sheet1', na_rep='''', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None)

Write Styler to an Excel sheet.

To write a single Styler to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an ExcelWriter object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique sheet_name. With all data written to the file it is necessary to save the changes. Note that creating an ExcelWriter object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

- **excel_writer**  [path-like, file-like, or ExcelWriter object]  File path or existing ExcelWriter.
- **sheet_name**  [str, default ‘Sheet1’]  Name of sheet which will contain DataFrame.
- **na_rep**  [str, default ‘’]  Missing data representation.
- **float_format**  [str, optional]  Format string for floating point numbers. For example float_format=’%.2f’ will format 0.1234 to 0.12.
- **columns**  [sequence or list of str, optional]  Columns to write.
- **header**  [bool or list of str, default True]  Write out the column names. If a list of string is given it is assumed to be aliases for the column names.
- **index**  [bool, default True]  Write row names (index).
- **index_label**  [str or sequence, optional]  Column label for index column(s) if desired. If not specified, and header and index are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- **startrow**  [int, default 0]  Upper left cell row to dump data frame.
- **startcol**  [int, default 0]  Upper left cell column to dump data frame.
- **engine**  [str, optional]  Write engine to use, ‘openpyxl’ or ‘xlsxwriter’. You can also set this via the options io.excel.xlsx.writer, io.excel.xls.writer, and io.excel.xlsxm.writer.
Deprecation since version 1.2.0: As the xlwt package is no longer maintained, the xlwt engine will be removed in a future version of pandas.

- **merge_cells** [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.
- **encoding** [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.
- **inf_rep** [str, default ‘inf’] Representation for infinity (there is no native representation for infinity in Excel).
- **verbose** [bool, default True] Display more information in the error logs.
- **freeze_panes** [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.
- **storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

See also:

- **to_csv** Write DataFrame to a comma-separated values (csv) file.
- **ExcelWriter** Class for writing DataFrame objects into excel sheets.
- **read_excel** Read an Excel file into a pandas DataFrame.
- **read_csv** Read a comma-separated values (csv) file into DataFrame.

**Notes**

For compatibility with to_csv(), to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

**Examples**

Create, write to and save a workbook:

```python
>>> df1 = pd.DataFrame([[‘a’, ‘b’], [‘c’, ‘d’]],
... index=[‘row 1’, ‘row 2’],
... columns=[‘col 1’, ‘col 2’])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```python
>>> df1.to_excel("output.xlsx",
... sheet_name=’Sheet name_1’)"
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:
```python
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
...    df1.to_excel(writer, sheet_name='Sheet_name_1')
...    df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:
```python
>>> with pd.ExcelWriter('output.xlsx', mode='a') as writer:
...    df.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the `engine` keyword (the default engine is automatically chosen depending on the file extension):
```python
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

### pandas.ExcelWriter

**class pandas.ExcelWriter**

```
ExcelWriter(path[, engine, date_format, ...])
```

Class for writing DataFrame objects into excel sheets.

**Parameters**

- **path** [str or typing.BinaryIO] Path to xls or xlsx or ods file.
- **engine** [str (optional)] Engine to use for writing. If None, defaults to io.excel.<extension>.
  
  **NOTE:** can only be passed as a keyword argument.

  Deprecated since version 1.2.0: As the xlwt package is no longer maintained, the xlwt engine will be removed in a future version of pandas.

- **date_format** [str, default None] Format string for dates written into Excel files (e.g. ‘YYYY-MM-DD’).
- **datetime_format** [str, default None] Format string for datetime objects written into Excel files.
  
  (e.g. ‘YYYY-MM-DD HH:MM:SS’).
- **mode** [{‘w’, ‘a’}, default ‘w’] File mode to use (write or append). Append does not work with fsspec URLs.
- **storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc., if using a URL that will be parsed by fsspec, e.g., starting “s3://”, “gcs://”.

  New in version 1.2.0.

- **if_sheet_exists** [{‘error’, ‘new’, ‘replace’, ‘overlay’}, default ‘error’] How to behave when trying to write to a sheet that already exists (append mode only).
  
  * error: raise a ValueError.
• new: Create a new sheet, with a name determined by the engine.
• replace: Delete the contents of the sheet before writing to it.
• overlay: Write contents to the existing sheet without removing the old contents.

New in version 1.3.0.

Changed in version 1.4.0: Added overlay option

**engine_kwargs** [dict, optional] Keyword arguments to be passed into the engine. These will be passed to the following functions of the respective engines:

- **xlsxwriter**: `xlsxwriter.Workbook(file, **engine_kwargs)`
- **openpyxl (write mode)**: `openpyxl.Workbook(**engine_kwargs)`
- **openpyxl (append mode)**: `openpyxl.load_workbook(file, **engine_kwargs)`
- **odswriter**: `odf.opendocument.OpenDocumentSpreadsheet(**engine_kwargs)`

New in version 1.3.0.

**kwargs** [dict, optional] Keyword arguments to be passed into the engine.

Deprecated since version 1.3.0: Use engine_kwargs instead.

Notes

None of the methods and properties are considered public.

For compatibility with CSV writers, ExcelWriter serializes lists and dicts to strings before writing.

Examples

Default usage:

```python
>>> df = pd.DataFrame(["ABC", "XYZ"], columns=["Foo", "Bar"])
>>> with pd.ExcelWriter("path_to_file.xlsx") as writer:
...     df.to_excel(writer)
```

To write to separate sheets in a single file:

```python
>>> df1 = pd.DataFrame(["AAA", "BBB"], columns=["Spam", "Egg"])
>>> df2 = pd.DataFrame(["ABC", "XYZ"], columns=["Foo", "Bar"])
>>> with pd.ExcelWriter("path_to_file.xlsx") as writer:
...     df1.to_excel(writer, sheet_name="Sheet1")
...     df2.to_excel(writer, sheet_name="Sheet2")
```

You can set the date format or datetime format:

```python
>>> from datetime import date, datetime
... >>> df = pd.DataFrame(...
...     [...
...         [date(2014, 1, 31), date(1999, 9, 24)],
...         [datetime(1998, 5, 26, 23, 33, 4), datetime(2014, 2, 28, 13, 5, 13)],
...     ],
...     index=["Date", "Datetime"],
...     columns=["X", "Y"],
... )
```
...)

```python
>>> with pd.ExcelWriter(
    ... 
    "path_to_file.xlsx",
    ... 
    date_format="YYYY-MM-DD",
    ... 
    datetime_format="YYYY-MM-DD HH:MM:SS"
    ... ) as writer:
    ... 
    df.to_excel(writer)
```

You can also append to an existing Excel file:

```python
>>> with pd.ExcelWriter("path_to_file.xlsx", mode="a", engine="openpyxl") as writer:
    ... 
    df.to_excel(writer, sheet_name="Sheet3")
```

Here, the `if_sheet_exists` parameter can be set to replace a sheet if it already exists:

```python
>>> with ExcelWriter(
    ... 
    "path_to_file.xlsx",
    ... 
    mode="a",
    ... 
    engine="openpyxl",
    ... 
    if_sheet_exists="replace",
    ... ) as writer:
    ... 
    df.to_excel(writer, sheet_name="Sheet1")
```

You can also write multiple DataFrames to a single sheet. Note that the `if_sheet_exists` parameter needs to be set to `overlay`:

```python
>>> with ExcelWriter("path_to_file.xlsx",
    ... 
    mode="a",
    ... 
    engine="openpyxl",
    ... 
    if_sheet_exists="overlay",
    ... ) as writer:
    ... 
    df1.to_excel(writer, sheet_name="Sheet1")
    ... 
    df2.to_excel(writer, sheet_name="Sheet1", startcol=3)
```

You can store Excel file in RAM:

```python
>>> import io
>>> df = pd.DataFrame(["ABC", "XYZ"], columns=["Foo", "Bar"])
>>> buffer = io.BytesIO()
>>> with pd.ExcelWriter(buffer) as writer:
    ... 
    df.to_excel(writer)
```

You can pack Excel file into zip archive:

```python
>>> import zipfile
>>> df = pd.DataFrame(["ABC", "XYZ"], columns=["Foo", "Bar"])
>>> with zipfile.ZipFile("path_to_file.zip", "w") as zf:
    ... 
    with zf.open("filename.xlsx", "w") as buffer:
    ... 
    with pd.ExcelWriter(buffer) as writer:
    ... 
    df.to_excel(writer)
```

You can specify additional arguments to the underlying engine:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> with pd.ExcelWriter(
...     "path_to_file.xlsx",
...     engine="xlsxwriter",
...     engine_kwargs={"options": {"nan_inf_to_errors": True}}
... ) as writer:
...     df.to_excel(writer)
```

In append mode, engine_kwargs are passed through to openpyxl's load_workbook:

```python
>>> with pd.ExcelWriter(
...     "path_to_file.xlsx",
...     engine="openpyxl",
...     mode="a",
...     engine_kwargs={"keep_vba": True}
... ) as writer:
...     df.to_excel(writer, sheet_name="Sheet2")
```

Attributes

| None |

Methods

| None |

3.1.5 JSON

| pandas.read_json(path_or_buf=None, orient=None, typ='frame', dtype=None, convert_axes=None, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, encoding_errors='strict', lines=False, chunksize=None, compression='infer', nrows=None, storage_options=None) Convert a JSON string to pandas object. |
| json_normalize(data, record_path, meta, ...) Normalizes semi-structured JSON data into a flat table. |
| DataFrame.to_json(path_or_buf=None, orient=None) Convert the object to a JSON string. |

Convert a JSON string to pandas object.

Parameters

path_or_buf [a valid JSON str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.json.

If you want to pass in a path object, pandas accepts any os.PathLike.
By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via `builtin open` function) or `StringIO`.

**orient** [str] Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:

- 'split': dict like `{index -> [index], columns -> [columns], data -> [values]}`
- 'records': list like `[{column -> value}, ..., {column -> value}]`
- 'index': dict like `{index -> {column -> value}}`
- 'columns': dict like `{column -> {index -> value}}`
- 'values': just the values array

The allowed and default values depend on the value of the `typ` parameter.

- when `typ` == 'series',
  - allowed orients are {'split', 'records', 'index'}
  - default is 'index'
  - The Series index must be unique for orient 'index'.

- when `typ` == 'frame',
  - allowed orients are {'split', 'records', 'index', 'columns', 'values', 'table'}
  - default is 'columns'
  - The DataFrame index must be unique for orients 'index' and 'columns'.
  - The DataFrame columns must be unique for orients 'index', 'columns', and 'records'.

**typ** [{'frame', 'series'}, default 'frame'] The type of object to recover.

**dtype** [bool or dict, default None] If True, infer dtypes; if a dict of column to dtype, then use those; if False, then don’t infer dtypes at all, applies only to the data.

For all orient values except 'table', default is True.

Changed in version 0.25.0: Not applicable for orient='table'.

**convert_axes** [bool, default None] Try to convert the axes to the proper dtypes.

For all orient values except 'table', default is True.

Changed in version 0.25.0: Not applicable for orient='table'.

**convert_dates** [bool or list of str, default True] If True then default datelike columns may be converted (depending on `keep_default_dates`). If False, no dates will be converted. If a list of column names, then those columns will be converted and default datelike columns may also be converted (depending on `keep_default_dates`).

**keep_default_dates** [bool, default True] If parsing dates (convert_dates is not False), then try to parse the default datelike columns. A column label is datelike if

- it ends with '_at',
- it ends with '_time',
- it begins with 'timestamp',
• it is 'modified', or
• it is 'date'.

**numpy** [bool, default False] Direct decoding to numpy arrays. Supports numeric data only, but non-numeric column and index labels are supported. Note also that the JSON ordering MUST be the same for each term if numpy=True.

Deprecated since version 1.0.0.

**precise_float** [bool, default False] Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (False) is to use fast but less precise built-in functionality.

**date_unit** [str, default None] The timestamp unit to detect if converting dates. The default behaviour is to try and detect the correct precision, but if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force parsing only seconds, milliseconds, microseconds or nanoseconds respectively.

**encoding** [str, default is 'utf-8'] The encoding to use to decode py3 bytes.

**encoding_errors** [str, optional, default “strict”] How encoding errors are treated. List of possible values.

New in version 1.3.0.

**lines** [bool, default False] Read the file as a json object per line.

**chunksize** [int, optional] Return JsonReader object for iteration. See the line-delimited json docs for more information on chunksize. This can only be passed if lines=True. If this is None, the file will be read into memory all at once.

Changed in version 1.2: JsonReader is a context manager.

**compression** [str or dict, default ‘infer’] For on-the-fly decompression of on-disk data. If ‘infer’ and ‘path_or_buf’ is path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key 'method' set to one of ['zip', 'gzip', 'bz2', 'zstd'] and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard. ZstdDecompressor, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: compression={"method": 'zstd', 'dict_data': my_compression_dict}.

Changed in version 1.4.0: Zstandard support.

**nrows** [int, optional] The number of lines from the line-delimited json file that has to be read. This can only be passed if lines=True. If this is None, all the rows will be returned.

New in version 1.1.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

**Returns**

* Series or DataFrame The type returned depends on the value of typ.

See also:
**DataFrame.to_json**  Convert a DataFrame to a JSON string.

**Series.to_json**  Convert a Series to a JSON string.

**json_normalize**  Normalize semi-structured JSON data into a flat table.

### Notes

Specific to orient='table', if a DataFrame with a literal Index name of index gets written with to_json(), the subsequent read operation will incorrectly set the Index name to None. This is because index is also used by DataFrame.to_json() to denote a missing Index name, and the subsequent read_json() operation cannot distinguish between the two. The same limitation is encountered with a MultiIndex and any names beginning with 'level_'.

### Examples

```python
df = pd.DataFrame([['a', 'b'], ['c', 'd']])
... index=['row 1', 'row 2'],
... columns=['col 1', 'col 2'])
```

Encoding/decoding a Dataframe using 'split' formatted JSON:

```python
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],"index":["row 1","row 2"],"data":[["a","b"],["c","d"]]}'

>>> pd.read_json(_, orient='split')
col 1 col 2
row 1 a b
row 2 c d
```

Encoding/decoding a Dataframe using 'index' formatted JSON:

```python
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'

>>> pd.read_json(_, orient='index')
col 1 col 2
row 1 a b
row 2 c d
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.

```python
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]

>>> pd.read_json(_, orient='records')
col 1 col 2
0 a b
1 c d
```

Encoding with Table Schema

3.1. Input/output
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.to_json(orient='table')

"{"schema":{"fields":[{"name":"index","type":"string"},{"name":"col 1","type":
    "string"},{"name":"col 2","type":"string"}],"primaryKey":"index"},"pandas_version":
    "1.4.0"},"data":{[{"index":"row 1","col 1":"a","col 2":"b"},{"index":"row 2",
    "col 1":"c","col 2":"d"}]}
```

pandas.json_normalize

```python
pandas.json_normalize(data, record_path=None, meta=None, meta_prefix=None, record_prefix=None,
errors='raise', sep='.', max_level=None)
```

Normalize semi-structured JSON data into a flat table.

**Parameters**

- `data` [dict or list of dicts] Unserialized JSON objects.
- `record_path` [str or list of str, default None] Path in each object to list of records. If not passed, data will be assumed to be an array of records.
- `meta` [list of paths (str or list of str), default None] Fields to use as metadata for each record in resulting table.
- `meta_prefix` [str, default None] If True, prefix records with dotted (?) path, e.g. foo.bar.field if meta is ['foo', 'bar'].
- `record_prefix` [str, default None] If True, prefix records with dotted (?) path, e.g. foo.bar.field if path to records is ['foo', 'bar'].
- `errors` [{'raise', 'ignore'}, default 'raise'] Configures error handling.
  - ‘ignore’ : will ignore KeyError if keys listed in meta are not always present.
  - ‘raise’ : will raise KeyError if keys listed in meta are not always present.
- `sep` [str, default ‘.’] Nested records will generate names separated by sep. e.g., for sep=’.’, {‘foo’: {‘bar’: 0}} -> foo.bar.
- `max_level` [int, default None] Max number of levels(depth of dict) to normalize. If None, normalizes all levels.

New in version 0.25.0.

**Returns**

- `frame` [DataFrame]

Normalize semi-structured JSON data into a flat table.

**Examples**

```python
>>> data = [
...     {"id": 1, "name": {"first": "Coleen", "last": "Volk"}},
...     {"name": {"given": "Mark", "family": "Regner"}},
...     {"id": 2, "name": "Faye Raker"},
... ]
```

```python
>>> pd.json_normalize(data)

<table>
<thead>
<tr>
<th>id</th>
<th>name.first</th>
<th>name.last</th>
<th>name</th>
<th>name</th>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>Coleen</td>
<td>Volk</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

1 NaN NaN NaN Mark Regner NaN
2 2.0 NaN NaN NaN NaN Faye Raker

>>> data = [
...     {
...         "id": 1,
...         "name": "Cole Volk",
...         "fitness": {"height": 130, "weight": 60},
...     },
...     {
...         "name": "Mark Reg", "fitness": {"height": 130, "weight": 60}},
...     {
...         "id": 2,
...         "name": "Faye Raker",
...         "fitness": {"height": 130, "weight": 60},
...     },
... ]

>>> pd.json_normalize(data, max_level=0)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cole Volk</td>
<td>{'height': 130, 'weight': 60}</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>Mark Reg</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>Faye Raker</td>
</tr>
</tbody>
</table>

Normalizes nested data up to level 1.

>>> data = [
...     {
...         "id": 1,
...         "name": "Cole Volk",
...         "fitness": {"height": 130, "weight": 60},
...     },
...     {
...         "name": "Mark Reg", "fitness": {"height": 130, "weight": 60}},
...     {
...         "id": 2,
...         "name": "Faye Raker",
...         "fitness": {"height": 130, "weight": 60},
...     },
... ]

>>> pd.json_normalize(data, max_level=1)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>fitness.height</th>
<th>fitness.weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cole Volk</td>
<td>130</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>Mark Reg</td>
<td>130</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>Faye Raker</td>
<td>130</td>
</tr>
</tbody>
</table>

>>> data = [
...     {
...         "state": "Florida",
...         "shortname": "FL",
...         "info": {"governor": "Rick Scott"},
...         "counties": [
...             {"name": "Dade", "population": 12345},
...             {"name": "Broward", "population": 400000},
...             {"name": "Palm Beach", "population": 600000},
...         ]
...     }
... ]

(continues on next page)
... },
... },
... {
... "state": "Ohio",
... "shortname": "OH",
... "info": {"governor": "John Kasich"},
... "counties": [
... {"name": "Summit", "population": 1234},
... {"name": "Cuyahoga", "population": 1337},
... ],
... },
... ]
>>> result = pd.json_normalize(
... data, "counties", ["state", "shortname", ["info", "governor"]]
... )
>>> result

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>state</th>
<th>shortname</th>
<th>info.governor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dade</td>
<td>12345</td>
<td>Florida</td>
<td>FL</td>
<td>Rick Scott</td>
</tr>
<tr>
<td>Broward</td>
<td>40000</td>
<td>Florida</td>
<td>FL</td>
<td>Rick Scott</td>
</tr>
<tr>
<td>Palm Beach</td>
<td>60000</td>
<td>Florida</td>
<td>FL</td>
<td>Rick Scott</td>
</tr>
<tr>
<td>Summit</td>
<td>1234</td>
<td>Ohio</td>
<td>OH</td>
<td>John Kasich</td>
</tr>
<tr>
<td>Cuyahoga</td>
<td>1337</td>
<td>Ohio</td>
<td>OH</td>
<td>John Kasich</td>
</tr>
</tbody>
</table>

>>> data = {"A": [1, 2]}
>>> pd.json_normalize(data, "A", record_prefix="Prefix.")

| Prefix.0
|---|
| 0 1
| 1 2

Returns normalized data with columns prefixed with the given string.

**pandas.DataFrame.to_json**

Dataframe.to_json(path_or_buf=None, orient=None, date_format=None, double_precision=10,
force_ascii=True, date_unit='ms', default_handler=None, lines=False, compression='infer',
index=True, indent=None, storage_options=None)

Convert the object to a JSON string.

Note NaN’s and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters**

- **path_or_buf** [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string.
- **orient** [str] Indication of expected JSON string format.
  - Series:
    - default is ‘index’
  - allowed values are: {'split', 'records', 'index', 'table'}.
  - DataFrame:
- default is ‘columns’
- allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.

The format of the JSON string:
- ‘split’ : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- ‘records’ : list like [{column -> value}, ... , {column -> value}]
- ‘index’ : dict like {index -> {column -> value}}
- ‘columns’ : dict like {column -> {index -> value}}
- ‘values’ : just the values array
- ‘table’ : dict like {'schema': {schema}, 'data': {data}}

Describing the data, where data component is like orient='records'.

**date_format**  [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the orient. For orient='table', the default is ‘iso’. For all other orients, the default is ‘epoch’.

**double_precision**  [int, default 10] The number of decimal places to use when encoding floating point values.

**force_ascii**  [bool, default True] Force encoded string to be ASCII.

**date_unit**  [str, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

**default_handler**  [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

**lines**  [bool, default False] If ‘orient’ is ‘records’ write out line-delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list-like.

**compression**  [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘path_or_buf’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

Changed in version 1.4.0: Zstandard support.

**index**  [bool, default True] Whether to include the index values in the JSON string. Not including the index (index=False) is only supported when orient is ‘split’ or ‘table’.

**indent**  [int, optional] Length of whitespace used to indent each record.

New in version 1.0.0.

**storage_options**  [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.
Returns

```
None or str  If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.
```

See also:

*read_json*  Convert a JSON string to pandas object.

Notes

The behavior of *indent=0* varies from the stdlib, which does not indent the output but does insert newlines. Currently, *indent=0* and the default *indent=None* are equivalent in pandas, though this may change in a future release.

*orient='table'* contains a ‘pandas_version’ field under ‘schema’. This stores the version of *pandas* used in the latest revision of the schema.

Examples

```python
>>> import json
>>> df = pd.DataFrame(
...     [["a", "b"], ["c", "d"]],
...     index=["row 1", "row 2"],
...     columns=["col 1", "col 2"],
... )

>>> result = df.to_json(orient="split")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "columns": ["col 1", "col 2"],
    "index": ["row 1", "row 2"],
    "data": [
        ["a", "b"],
        ["c", "d"],
    ]
}
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.
Encoding/decoding a Dataframe using 'index' formatted JSON:

```python
>>> result = df.to_json(orient="index")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
        "col 2": "d"
    }
}
```

Encoding/decoding a Dataframe using 'columns' formatted JSON:

```python
>>> result = df.to_json(orient="columns")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "col 1": {
        "row 1": "a",
        "row 2": "c"
    },
    "col 2": {
        "row 1": "b",
        "row 2": "d"
    }
}
```

Encoding/decoding a Dataframe using 'values' formatted JSON:

```python
>>> result = df.to_json(orient="values")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
[
    [
        "col 1": "a",
        "col 2": "b"
    ],
    [
        "col 1": "c",
        "col 2": "d"
    ]
]
```
Encoding with Table Schema:

```python
>>> result = df.to_json(orient="table")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "schema": {
        "fields": [
            {
                "name": "index",
                "type": "string"
            },
            {
                "name": "col 1",
                "type": "string"
            },
            {
                "name": "col 2",
                "type": "string"
            }
        ],
        "primaryKey": ["index"],
        "pandas_version": "1.4.0"
    },
    "data": [
        {"index": "row 1",
         "col 1": "a",
         "col 2": "b"
        },
        {"index": "row 2",
         "col 1": "c",
         "col 2": "d"
        }
    ]
}
```

`build_table_schema(data[, index, ...])` Create a Table schema from `data`.
pandas.io.json.build_table_schema

Create a Table schema from data.

**Parameters**

- **data** [Series, DataFrame]
- **index** [bool, default True] Whether to include data.index in the schema.
- **primary_key** [bool or None, default True] Column names to designate as the primary key. The default None will set 'primaryKey' to the index level or levels if the index is unique.
- **version** [bool, default True] Whether to include a field pandas_version with the version of pandas that last revised the table schema. This version can be different from the installed pandas version.

**Returns**

- **schema** [dict]

**Notes**

See Table Schema for conversion types. Timedeltas as converted to ISO8601 duration format with 9 decimal places after the seconds field for nanosecond precision.

Categoricals are converted to the any dtype, and use the enum field constraint to list the allowed values. The ordered attribute is included in an ordered field.

**Examples**

```python
>>> df = pd.DataFrame(
...     {'A': [1, 2, 3],
...      'B': ['a', 'b', 'c'],
...      'C': pd.date_range('2016-01-01', freq='d', periods=3),
...     }, index=pd.Index(range(3), name='idx'))

>>> build_table_schema(df)

{'fields': [{'name': 'idx', 'type': 'integer'}, {'name': 'A', 'type': 'integer'}, {
    'name': 'B', 'type': 'string'}, {'name': 'C', 'type': 'datetime'}], 'primaryKey': ['idx'], 'pandas_version': '1.4.0'}
```

### 3.1.6 HTML

- `read_html(io[, match, flavor, header, ...])` Read HTML tables into a list of DataFrame objects.
- `DataFrame.to_html([buf, columns, col_space, ...])` Render a DataFrame as an HTML table.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.read_html

pandas.read_html(io=None, match=None, flavor=None, header=None, index_col=None, skiprows=None, attrs=None, parse_dates=False, thousands=None, encoding=None, decimal=None, converters=None, na_values=None, keep_default_na=True, displayed_only=True)

Read HTML tables into a list of DataFrame objects.

Parameters

- **io** [str, path object, or file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a string read() function. The string can represent a URL or the HTML itself. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with 'https' you might try removing the 's'.

- **match** [str or compiled regular expression, optional] The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to '.+' (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

- **flavor** [str, optional] The parsing engine to use. ‘bs4’ and ‘html5lib’ are synonymous with each other, they are both there for backwards compatibility. The default of None tries to use lxml to parse and if that fails it falls back on bs4 + html5lib.

- **header** [int or list-like, optional] The row (or list of rows for a MultiIndex) to use to make the columns headers.

- **index_col** [int or list-like, optional] The column (or list of columns) to use to create the index.

- **skiprows** [int, list-like or slice, optional] Number of rows to skip after parsing the column integer. 0-based. If a sequence of integers or a slice is given, will skip the rows indexed by that sequence. Note that a single element sequence means ‘skip the nth row’ whereas an integer means ‘skip n rows’.

- **attrs** [dict, optional] This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to lxml or Beautiful Soup. However, these attributes must be valid HTML table attributes to work correctly. For example,

  ```
  attrs = {'id': 'table'}
  ```

  is a valid attribute dictionary because the ‘id’ HTML tag attribute is a valid HTML attribute for any HTML tag as per this document.

  ```
  attrs = {'asdf': 'table'}
  ```

  is not a valid attribute dictionary because ‘asdf’ is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found here. A working draft of the HTML 5 spec can be found here. It contains the latest information on table attributes for the modern web.

- **parse_dates** [bool, optional] See read_csv() for more details.

- **thousands** [str, optional] Separator to use to parse thousands. Defaults to ',', '.

- **encoding** [str, optional] The encoding used to decode the web page. Defaults to None. "None" preserves the previous encoding behavior, which depends on the underlying parser library (e.g., the parser library will try to use the encoding provided by the document).
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

**decimal** [str, default ‘.’] Character to recognize as decimal point (e.g. use ‘,’ for European data).

**converters** [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the cell (not column) content, and return the transformed content.

**na_values** [iterable, default None] Custom NA values.

**keep_default_na** [bool, default True] If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they’re appended to.

**displayed_only** [bool, default True] Whether elements with “display: none” should be parsed.

**Returns**

dfs A list of DataFrames.

**See also:**

`read_csv` Read a comma-separated values (csv) file into DataFrame.

**Notes**

Before using this function you should read the *gotchas about the HTML parsing libraries*.

Expect to do some cleanup after you call this function. For example, you might need to manually assign column names if the column names are converted to NaN when you pass the header=0 argument. We try to assume as little as possible about the structure of the table and push the idiosyncrasies of the HTML contained in the table to the user.

This function searches for `<table>` elements and only for `<tr>` and `<th>` rows and `<td>` elements within each `<tr>` or `<th>` element in the table. `<td>` stands for “table data”. This function attempts to properly handle colspan and rowspan attributes. If the function has a `<thead>` argument, it is used to construct the header, otherwise the function attempts to find the header within the body (by putting rows with only `<th>` elements into the header).

Similar to `read_csv()` the header argument is applied after skiprows is applied.

This function will always return a list of `DataFrame` or it will fail, e.g., it will not return an empty list.

**Examples**

See the *read_html documentation in the IO section of the docs* for some examples of reading in HTML tables.

**pandas.DataFrame.to_html**

```
DataFrame.to_html(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, max_rows=None, max_cols=None, show_dimensions=False, decimal='.', bold_rows=True, classes=None, escape=True, notebook=False, border=None, table_id=None, render_links=False, encoding=None)
```

Render a DataFrame as an HTML table.

**Parameters**

- **buf** [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.
columns [sequence, optional, default None] The subset of columns to write. Writes all columns by default.

col_space [str or int, list or dict of int or str, optional] The minimum width of each column in CSS length units. An int is assumed to be px units.

New in version 0.25.0: Ability to use str.

header [bool, optional] Whether to print column labels, default True.

index [bool, optional, default True] Whether to print index (row) labels.

na_rep [str, optional, default ‘NaN’] String representation of NaN to use.

formatters [list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns’ elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format [one-parameter function, optional, default None] Formatter function to apply to columns’ elements if they are floats. This function must return a unicode string and will be applied only to the non-NaN elements, with NaN being handled by na_rep.

Changed in version 1.2.0.

sparsify [bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names [bool, optional, default True] Prints the names of the indexes.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows [int, optional] Maximum number of rows to display in the console.

max_cols [int, optional] Maximum number of columns to display in the console.

show_dimensions [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal [str, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.

bold_rows [bool, default True] Make the row labels bold in the output.

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

escape [bool, default True] Convert the characters <, >, and & to HTML-safe sequences.
notebook  [’True, False], default False] Whether the generated HTML is for IPython Notebook.

border  [int] A border=border attribute is included in the opening <table> tag. Default pd.options.display.html.border.

table_id  [str, optional] A css id is included in the opening <table> tag if specified.

render_links  [bool, default False] Convert URLs to HTML links.

encoding  [str, default “utf-8”] Set character encoding.

New in version 1.0.

Returns

str or None  If buf is None, returns the result as a string. Otherwise returns None.

See also:

to_string  Convert DataFrame to a string.

<table>
<thead>
<tr>
<th>Styler.to_html(buf, table_uuid, ...)</th>
<th>Write Styler to a file, buffer or string in HTML-CSS format.</th>
</tr>
</thead>
</table>

pandas.io.formats.style.Styler.to_html

Styler.to_html(buf=None, *, table_uuid=None, table_attributes=None, sparse_index=None, sparse_columns=None, bold_headers=False, caption=None, max_rows=None, max_columns=None, encoding=None, doctype_html=False, exclude_styles=False, **kwargs)

Write Styler to a file, buffer or string in HTML-CSS format.

New in version 1.3.0.

Parameters

buf  [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a string write() function. If None, the result is returned as a string.

table_uuid  [str, optional] Id attribute assigned to the <table> HTML element in the format:

<table id="T_<table_uuid>" ..>

If not given uses Styler’s initially assigned value.

table_attributes  [str, optional] Attributes to assign within the <table> HTML element in the format:

<table .. <table_attributes> >

If not given defaults to Styler’s preexisting value.

sparse_index  [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. Defaults to pandas.options.styler.sparse.index value.

New in version 1.4.0.

sparse_columns  [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each column. Defaults to pandas.options.styler.sparse.columns value.
**bold_headers** [bool, optional] Adds “font-weight: bold;” as a CSS property to table style header cells.

   New in version 1.4.0.

**caption** [str, optional] Set, or overwrite, the caption on Styler before rendering.

   New in version 1.4.0.

**max_rows** [int, optional] The maximum number of rows that will be rendered. Defaults to pandas.options.styler.render.max_rows/max_columns.

   New in version 1.4.0.

**max_columns** [int, optional] The maximum number of columns that will be rendered. Defaults to pandas.options.styler.render.max_columns, which is None.

   Rows and columns may be reduced if the number of total elements is large. This value is set to pandas.options.styler.render.max_elements, which is 262144 (18 bit browser rendering).

   New in version 1.4.0.

**encoding** [str, optional] Character encoding setting for file output, and HTML meta tags. Defaults to pandas.options.styler.render.encoding value of “utf-8”.

**doctype_html** [bool, default False] Whether to output a fully structured HTML file including all HTML elements, or just the core <style> and <table> elements.

**exclude_styles** [bool, default False] Whether to include the <style> element and all associated element class and id identifiers, or solely the <table> element without styling identifiers.

**kwargs** Any additional keyword arguments are passed through to the jinja2 self.
.template.render process. This is useful when you need to provide additional variables for a custom template.

**Returns**

   str or None  If buf is None, returns the result as a string. Otherwise returns None.

**See also:**

- **DataFrame.to_html**  Write a DataFrame to a file, buffer or string in HTML format.

### 3.1.7 XML

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_xml</td>
<td>Read XML document into a DataFrame object.</td>
</tr>
<tr>
<td>DataFrame.to_xml</td>
<td>Render a DataFrame to an XML document.</td>
</tr>
</tbody>
</table>

1074 Chapter 3.  API reference
pandas.read_xml

Read XML document into a DataFrame object.

New in version 1.3.0.

Parameters

- **path_or_buffer** [str, path object, or file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a read() function. The string can be any valid XML string or a path. The string can further be a URL. Valid URL schemes include http, ftp, s3, and file.

- **xpath** [str, optional, default './*'] The XPath to parse required set of nodes for migration to DataFrame. XPath should return a collection of elements and not a single element. Note: The etree parser supports limited XPath expressions. For more complex XPath, use lxml which requires installation.

- **namespaces** [dict, optional] The namespaces defined in XML document as dicts with key being namespace prefix and value the URI. There is no need to include all namespaces in XML, only the ones used in xpath expression. Note: if XML document uses default namespace denoted as xmlns='<URI>' without a prefix, you must assign any temporary namespace prefix such as 'doc' to the URI in order to parse underlying nodes and/or attributes. For example,

  ```python
  namespaces = {'doc': 'https://example.com'}
  ```

- **elems_only** [bool, optional, default False] Parse only the child elements at the specified xpath. By default, all child elements and non-empty text nodes are returned.

- **attrs_only** [bool, optional, default False] Parse only the attributes at the specified xpath. By default, all attributes are returned.

- **names** [list-like, optional] Column names for DataFrame of parsed XML data. Use this parameter to rename original element names and distinguish same named elements.

- **encoding** [str, optional, default 'utf-8'] Encoding of XML document.

- **parser** [{lxml,etree}, default lxml] Parser module to use for retrieval of data. Only lxml and ‘etree’ are supported. With lxml more complex XPath searches and ability to use XSLT stylesheet are supported.

- **stylesheet** [str, path object or file-like object] A URL, file-like object, or a raw string containing an XSLT script. This stylesheet should flatten complex, deeply nested XML documents for easier parsing. To use this feature you must have lxml module installed and specify ‘lxml’ as parser. The xpath must reference nodes of transformed XML document generated after XSLT transformation and not the original XML document. Only XSLT 1.0 scripts and not later versions is currently supported.

- **compression** [str or dict, default ‘infer’] For on-the-fly decompression of on-disk data. If ‘infer’ and ‘path_or_buffer’ is path-like, then detect compression from the following extensions: .gz, .bz2, .zip, .xz, or .zst (otherwise no compression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key 'method' set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the follow-
ing could be passed for Zstandard decompression using a custom compression dictionary:

```python
compression={"method": "zstd", "dict_data": my_compression_dict}
```

Changed in version 1.4.0: Zstandard support.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

**Returns**

**df** A DataFrame.

**See also:**

- `read_json` Convert a JSON string to pandas object.
- `read_html` Read HTML tables into a list of DataFrame objects.

**Notes**

This method is best designed to import shallow XML documents in following format which is the ideal fit for the two-dimensions of a DataFrame (row by column).

```xml
<root>
  <row>
    <column1>data</column1>
    <column2>data</column2>
    <column3>data</column3>
    ...
  </row>
  <row>
    ...
  </row>
  ...
</root>
```

As a file format, XML documents can be designed any way including layout of elements and attributes as long as it conforms to W3C specifications. Therefore, this method is a convenience handler for a specific flatter design and not all possible XML structures.

However, for more complex XML documents, **stylesheet** allows you to temporarily redesign original document with XSLT (a special purpose language) for a flatter version for migration to a DataFrame.

This function will **always** return a single **DataFrame** or raise exceptions due to issues with XML document, xpath, or other parameters.
Examples

```python
>>> xml = '''<?xml version='1.0' encoding='utf-8'?>
... <data xmlns="http://example.com">
... <row>
...     <shape>square</shape>
...     <degrees>360</degrees>
...     <sides>4.0</sides>
... </row>
... <row>
...     <shape>circle</shape>
...     <degrees>360</degrees>
...     <sides/>
... </row>
... <row>
...     <shape>triangle</shape>
...     <degrees>180</degrees>
...     <sides>3.0</sides>
... </row>
... </data>'''
... 
>>> df = pd.read_xml(xml)
... 
>>> df
shape degrees sides
0   square    360   4.0
1   circle    360  NaN
2  triangle   180   3.0

>>> xml = '''<?xml version='1.0' encoding='utf-8'?>
... <data>
... <row shape="square" degrees="360" sides="4.0"/>
... <row shape="circle" degrees="360"/>
... <row shape="triangle" degrees="180" sides="3.0"/>
... </data>'''
... 
>>> df = pd.read_xml(xml, xpath=".//row")
... 
>>> df
shape degrees sides
0   square    360   4.0
1   circle    360  NaN
2  triangle   180   3.0

>>> xml = '''<?xml version='1.0' encoding='utf-8'?>
... <doc:data xmlns:doc="https://example.com">
... <doc:row>
...     <doc:shape>square</doc:shape>
...     <doc:degrees>360</doc:degrees>
...     <doc:sides>4.0</doc:sides>
... </doc:row>
... <doc:row>
...     <doc:shape>circle</doc:shape>
...     <doc:degrees>360</doc:degrees>
... </doc:row>
... </doc:data>

(continues on next page)
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

...
<doc:sides/>
...
</doc:row>
...
<doc:row>
...
<doc:shape>triangle</doc:shape>
...
<doc:degrees>180</doc:degrees>
...
<doc:sides>3.0</doc:sides>
...
</doc:row>
... </doc:data>'''
>>> df = pd.read_xml(xml,
...
xpath="//doc:row",
...
namespaces={"doc": "https://example.com"})
>>> df
shape degrees sides
0
square
360
4.0
1
circle
360
NaN
2 triangle
180
3.0

pandas.DataFrame.to_xml
DataFrame.to_xml(path_or_buffer=None, index=True, root_name='data', row_name='row', na_rep=None,
attr_cols=None, elem_cols=None, namespaces=None, prefix=None, encoding='utf-8',
xml_declaration=True, pretty_print=True, parser='lxml', stylesheet=None,
compression='infer', storage_options=None)
Render a DataFrame to an XML document.
New in version 1.3.0.
Parameters
path_or_buffer [str, path object, file-like object, or None, default None] String, path object
(implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string.
index [bool, default True] Whether to include index in XML document.
root_name [str, default ‘data’] The name of root element in XML document.
row_name [str, default ‘row’] The name of row element in XML document.
attr_cols [list-like, optional] List of columns to write as attributes in row element. Hierarchical columns will be flattened with underscore delimiting the different levels.
elem_cols [list-like, optional] List of columns to write as children in row element. By default,
all columns output as children of row element. Hierarchical columns will be flattened
with underscore delimiting the different levels.
namespaces [dict, optional] All namespaces to be defined in root element. Keys of dict should
be prefix names and values of dict corresponding URIs. Default namespaces should be
given empty string key. For example,
namespaces = {"": "https://example.com"}

1078

Chapter 3. API reference


prefix [str, optional] Namespace prefix to be used for every element and/or attribute in document. This should be one of the keys in namespaces dict.

encoding [str, default ‘utf-8’] Encoding of the resulting document.

xml_declaration [bool, default True] Whether to include the XML declaration at start of document.

pretty_print [bool, default True] Whether output should be pretty printed with indentation and line breaks.

canvas [[{'lxml'}, 'etree'], default {'lxml'}] Parser module to use for building of tree. Only ‘lxml’ and ‘etree’ are supported. With ‘lxml’, the ability to use XSLT stylesheet is supported.

stylesheet [str, path object or file-like object, optional] A URL, file-like object, or a raw string containing an XSLT script used to transform the raw XML output. Script should use layout of elements and attributes from original output. This argument requires lxml to be installed. Only XSLT 1.0 scripts and not later versions is currently supported.

compression [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘path_or_buffer’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of [{'zip', 'gzip', 'bz2', 'zstd'}] and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

Changed in version 1.4.0: Zstandard support.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

Returns

None or str If io is None, returns the resulting XML format as a string. Otherwise returns None.

See also:

to_json Convert the pandas object to a JSON string.
to_html Convert DataFrame to a html.

Examples

```python
>>> df = pd.DataFrame({'shape': ['square', 'circle', 'triangle'],
...                     'degrees': [360, 360, 180],
...                     'sides': [4, np.nan, 3]})

>>> df.to_xml()
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
```

(continues on next page)
<index>0</index>
<shape>square</shape>
<degrees>360</degrees>
<sides>4.0</sides>
</row>
<row>
<index>1</index>
<shape>circle</shape>
<degrees>360</degrees>
<sides/>
</row>
<row>
<index>2</index>
<shape>triangle</shape>
<degrees>180</degrees>
<sides>3.0</sides>
</row>
</data>

```python
>>> df.to_xml(attr_cols=['index', 'shape', 'degrees', 'sides'])
<?xml version='1.0' encoding='utf-8'>
<data>
   <row index="0" shape="square" degrees="360" sides="4.0"/>
   <row index="1" shape="circle" degrees="360"/>
   <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

```python
>>> df.to_xml(namespaces={'doc': 'https://example.com'},
            prefix='doc')
<?xml version='1.0' encoding='utf-8'>
<doc:data xmlns:doc="https://example.com">
   <doc:row>
      <doc:index>0</doc:index>
      <doc:shape>square</doc:shape>
      <doc:degrees>360</doc:degrees>
      <doc:sides>4.0</doc:sides>
   </doc:row>
   <doc:row>
      <doc:index>1</doc:index>
      <doc:shape>circle</doc:shape>
      <doc:degrees>360</doc:degrees>
      <doc:sides/>
   </doc:row>
   <doc:row>
      <doc:index>2</doc:index>
      <doc:shape>triangle</doc:shape>
      <doc:degrees>180</doc:degrees>
      <doc:sides>3.0</doc:sides>
   </doc:row>
</doc:data>
```
3.1.8 LaTeX

DataFrame.to_latex([buf, columns, ...])

Render object to a LaTeX tabular, longtable, or nested table.

pandas.DataFrame.to_latex

DataFrame.to_latex(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None, decimal=',', multicolumn=None, multicolumn_format=None, multirow=None, caption=None, label=None, position=None)

Render object to a LaTeX tabular, longtable, or nested table.

Requires \usepackage{booktabs}. The output can be copy/pasted into a main LaTeX document or read from an external file with \input{table.tex}.

Changed in version 1.0.0: Added caption and label arguments.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

Parameters

buf [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns [list of label, optional] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

na_rep [str, default ‘NaN’] Missing data representation.

formatters [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns’ elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function or str, optional, default None] Formatter for floating point numbers. For example float_format="%.2f" and float_format="{:0.2f}". This will both result in 0.1234 being formatted as 0.12.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names [bool, default True] Prints the names of the indexes.

bold_rows [bool, default False] Make the row labels bold in the output.

column_format [str, optional] The columns format as specified in LaTeX table format e.g. ‘rc’l for 3 columns. By default, ‘l’ will be used for all columns except columns of numbers, which default to ‘r’.

longtable [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a usepackage{longtable} to your LaTeX preamble.
escape [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to `utf-8`.

decimal [str, default `.'] Character recognized as decimal separator, e.g. `.' in Europe.

multicolumn [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

multicolumn_format [str, default `{}`] The alignment for multicolumns, similar to `\column_format` The default will be read from the config module.

multirow [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a `\usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

caption [str or tuple, optional] Tuple (full_caption, short_caption), which results in `\caption[short_caption]{full_caption}`; if a single string is passed, no short caption will be set.

New in version 1.0.0.

Changed in version 1.2.0: Optionally allow caption to be a tuple (full_caption, short_caption).

label [str, optional] The LaTeX label to be placed inside `\label{}` in the output. This is used with `\ref{}` in the main `.tex` file.

New in version 1.0.0.

position [str, optional] The LaTeX positional argument for tables, to be placed after `\begin{}` in the output.

New in version 1.2.0.

Returns

str or None If buf is None, returns the result as a string. Otherwise returns None.

See also:

`Styler.to_latex` Render a DataFrame to LaTeX with conditional formatting.

`DataFrame.to_string` Render a DataFrame to a console-friendly tabular output.

`DataFrame.to_html` Render a DataFrame as an HTML table.

Examples

```python
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                    mask=['red', 'purple'],
...                    weapon=['sai', 'bo staff']))

>>> print(df.to_latex(index=False))
\begin{tabular}{lll}
\toprule
name & mask & weapon \\
\midrule
Raphael & red & sai \\
\bottomrule
\end{tabular}
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

Donatello & purple & bo staff \bottomrule
\end{tabular}

Styler.to_latex([buf, column_format,...]) Write Styler to a file, buffer or string in LaTeX format.

pandas.io.formats.style.Styler.to_latex

Styler.to_latex(buf=None, *, column_format=None, position=None, position_float=None, hrules=None, clines=None, label=None, caption=None, sparse_index=None, sparse_columns=None, multrow_align=None, multicoll_align=None, siunitx=False, environment=None, encoding=None, convert_css=False)

Write Styler to a file, buffer or string in LaTeX format.

New in version 1.3.0.

Parameters

- **buf** [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a string write() function. If None, the result is returned as a string.

- **column_format** [str, optional] The LaTeX column specification placed in location:

  \begin{tabular}{<column_format>}

  Defaults to ‘l’ for index and non-numeric data columns, and, for numeric data columns, to ‘r’ by default, or ‘S’ if siunitx is True.

- **position** [str, optional] The LaTeX positional argument (e.g. ‘h!’) for tables, placed in location:

  \begin{table}[

- **position_float** [“centering”, “raggedleft”, “raggedright”), optional] The LaTeX float command placed in location:

  \begin{table}[

  Cannot be used if environment is “longtable”.

- **hrules** [bool] Set to True to add \toprule, \midrule and \bottomrule from the \{booktabs\} LaTeX package. Defaults to pandas.options.styler.latex.hrules, which is False.

  Changed in version 1.4.0.

- **clines** [str, optional] Use to control adding \cline commands for the index labels separation. Possible values are:

  - **None**: no cline commands are added (default).
  - **“all:data”**: a cline is added for every index value extending the width of the table, including data entries.
  - **“all:index”**: as above with lines extending only the width of the index entries.
  - **“skip-last:data”**: a cline is added for each index value except the last level (which is never sparsified), extending the width of the table.

3.1. Input/output 1083
• “skip-last;index”: as above with lines extending only the width of the index entries.
  New in version 1.4.0.

**label** [str, optional] The LaTeX label included as: \label{<label>}. This is used with \ref{<label>} in the main .tex file.

**caption** [str, tuple, optional] If string, the LaTeX table caption included as: \caption{<caption>}. If tuple, i.e (“full caption”, “short caption”), the caption included as: \caption[<caption[1]>][]{<caption[0]>}.

**sparse_index** [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. Defaults to pandas.options.styler.sparse.index, which is True.

**sparse_columns** [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each column. Defaults to pandas.options.styler.sparse.columns, which is True.

**multirow_align** [{“c”, “t”, “b”, “naive”}, optional] If sparsifying hierarchical MultiIndexes whether to align text centrally, at the top or bottom using the multirow package. If not given defaults to pandas.options.styler.latex.multirow_align, which is “c”. If “naive” is given renders without multirow.
  Changed in version 1.4.0.

**multicol_align** [{“r”, “c”, “l”, “naive-l”, “naive-r”}, optional] If sparsifying hierarchical MultiIndex columns whether to align text at the left, centrally, or at the right. If not given defaults to pandas.options.styler.latex.multicol_align, which is “r”. If a naive option is given renders without multicol. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. “|r” will draw a rule on the left side of right aligned merged cells.
  Changed in version 1.4.0.

**siunitx** [bool, default False] Set to True to structure LaTeX compatible with the \{siunitx\} package.

**environment** [str, optional] If given, the environment that will replace ‘table’ in \begin{table}. If ‘longtable’ is specified then a more suitable template is rendered. If not given defaults to pandas.options.styler.latex.environment, which is None.
  New in version 1.4.0.

**encoding** [str, optional] Character encoding setting. Defaults to pandas.options.styler.render.encoding, which is “utf-8”.

**convert_css** [bool, default False] Convert simple cell-styles from CSS to LaTeX format. Any CSS not found in conversion table is dropped. A style can be forced by adding option –latex. See notes.

**Returns**

- **str or None** If buf is None, returns the result as a string. Otherwise returns None.

**See also:**

Styler.format Format the text display value of cells.
Notes

LaTeX Packages

For the following features we recommend the following \LaTeX{} inclusions:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse columns</td>
<td>none: included within default {tabular} environment</td>
</tr>
<tr>
<td>sparse rows</td>
<td>\usepackage{multirow}</td>
</tr>
<tr>
<td>hrules</td>
<td>\usepackage{booktabs}</td>
</tr>
<tr>
<td>colors</td>
<td>\usepackage{table}{xcolor}</td>
</tr>
<tr>
<td>siunitx</td>
<td>\usepackage{siunitx}</td>
</tr>
<tr>
<td>bold (with siunitx)</td>
<td>\usepackage{etoolbox}</td>
</tr>
<tr>
<td></td>
<td>\robustify\bfseries</td>
</tr>
<tr>
<td></td>
<td>\sisetup{detect-all = true} {within {document}}</td>
</tr>
<tr>
<td>italic (with siunitx)</td>
<td>\usepackage{etoolbox}</td>
</tr>
<tr>
<td></td>
<td>\robustify\itshape</td>
</tr>
<tr>
<td></td>
<td>\sisetup{detect-all = true} {within {document}}</td>
</tr>
<tr>
<td>environment</td>
<td>\usepackage{longtable} if arg is “longtable”</td>
</tr>
<tr>
<td>hyperlinks</td>
<td>\usepackage{hyperref}</td>
</tr>
</tbody>
</table>

Cell Styles

\LaTeX{} styling can only be rendered if the accompanying styling functions have been constructed with appropriate \LaTeX{} commands. All styling functionality is built around the concept of a CSS \(<attribute>, <value>\) pair (see Table Visualization), and this should be replaced by a \LaTeX{} \(<command>, <options>\) approach. Each cell will be styled individually using nested \LaTeX{} commands with their accompanied options.

For example the following code will highlight and bold a cell in HTML-CSS:

```python
>>> df = pd.DataFrame([[1,2], [3,4]])
>>> s = df.style.highlight_max(axis=None,
...                             props='background-color:red; font-weight:bold;')
>>> s.to_html()
```

The equivalent using \LaTeX{} only commands is the following:

```python
>>> s = df.style.highlight_max(axis=None,
...                             props='cellcolor:{red}; bfseries: ;')
>>> s.to_latex()
```

Internally these structured \LaTeX{} \(<command>, <options>\) pairs are translated to the display_value with the default structure: \(<command><options> <display_value>\). Where there are multiple commands the latter is nested recursively, so that the above example highlighted cell is rendered as \cellcolor{red} \bfseries 4.

Occasionally this format does not suit the applied command, or combination of \LaTeX{} packages that is in use, so additional flags can be added to the \<options\>, within the tuple, to result in different positions of required braces (the default being the same as --nowrap):

3.1. Input/output
For example the `textbf` command for font-weight should always be used with `--rwrap` so (`'textbf', '--rwrap'`) will render a working cell, wrapped with braces, as `\textbf{<display_value>}`.

A more comprehensive example is as follows:

```python
>>> df = pd.DataFrame([[1, 2.2, "dogs"], [3, 4.4, "cats"], [2, 6.6, "cows"]],
    index=["ix1", "ix2", "ix3"],
    columns=["Integers", "Floats", "Strings"])
>>> s = df.style.highlight_max(
    props="cellcolor:[HTML]{FFFF00}; color:{red};";
    'textit:--rwrap; textbf:--rwrap;
    ...)
>>> s.to_latex()

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ix1</td>
<td>1</td>
<td>2.200000</td>
</tr>
<tr>
<td>ix2</td>
<td>3</td>
<td>4.400000</td>
</tr>
<tr>
<td>ix3</td>
<td>2</td>
<td>6.600000</td>
</tr>
</tbody>
</table>
```

**Table Styles**

Internally Styler uses its `table_styles` object to parse the `column_format`, `position`, `position_float`, and `label` input arguments. These arguments are added to table styles in the format:

```python
set_table_styles([
    {"selector": "column_format", "props": f':{column_format};"},
    {"selector": "position", "props": f':{position};"},
    {"selector": "position_float", "props": f':{position_float};"},
    {"selector": "label", "props": f'{{(label.replace(':','§'))}}';"}
], overwrite=False)
```

If other commands are added to table styles they will be detected, and positioned immediately above the `\begin{tabular}` command. For example to add odd and even row coloring, from the `{colortbl}` package, in format `\rowcolors{1}{pink}{red}`, use:

```python
set_table_styles([
    {'selector': 'toprule', 'props': ':toprule;'},
    {'selector': 'bottomrule', 'props': ':hline;'},
], overwrite=False)
```
```python
set_table_styles([  
    {'selector': 'rowcolors', 'props': ':{1}{pink}{red};'}
], overwrite=False)
```

A more comprehensive example using these arguments is as follows:

```python
>>> df.columns = pd.MultiIndex.from_tuples([  
...     ("Numeric", "Integers"),
...     ("Numeric", "Floats"),
...     ("Non-Numeric", "Strings")
... ])
>>> df.index = pd.MultiIndex.from_tuples([  
...     ("L0", "ix1"), ("L0", "ix2"), ("L1", "ix3")
... ])
>>> s = df.style.highlight_max(  
...     props='cellcolor:[HTML]{FFFF00}; color:red; itshape:; bfseries:'
... )
>>> s.to_latex(  
...     column_format="rrrrr", position="h", position_float="centering",  
...     hrules=True, label="table:5", caption="Styled LaTeX Table",  
...     multirow_align="t", multicol_align="r"
... )
```

![Table 1: Styled LaTeX Table](image)

### Formatting

To format values `Styler.format()` should be used prior to calling `Styler.to_latex`, as well as other methods such as `Styler.hide()` for example:

```python
>>> s.clear()
>>> s.table_styles = []
>>> s.caption = None
>>> s.format({  
...     ("Numeric", "Integers"): '$\{}$',  
...     ("Numeric", "Floats"): '{:.3f}',  
...     ("Non-Numeric", "Strings"): str.upper
... })
```

```latex
\begin{tabular}{lrrl}
  L0 & ix1 & 1 & 2.20000 \textcolor{red}{dogs} \\
  ix2 & 3 & 4.40000 & cats \\
  L1 & ix3 & 2 & 6.60000 \textcolor{red}{cows} \\
\end{tabular}
```

```python
>>> s.to_latex()
```

(continues on next page)
CSS Conversion

This method can convert a Styler constructed with HTML-CSS to LaTeX using the following limited conversions.

<table>
<thead>
<tr>
<th>CSS Attribute</th>
<th>CSS value</th>
<th>LaTeX Command</th>
<th>LaTeX Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>font-weight</td>
<td>bold</td>
<td>bfseries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bolder</td>
<td>bfseries</td>
<td></td>
</tr>
<tr>
<td>font-style</td>
<td>italic</td>
<td>itshape</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oblique</td>
<td>slshape</td>
<td></td>
</tr>
<tr>
<td>background-color</td>
<td>red</td>
<td>cellcolor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#fe01ea</td>
<td></td>
<td>{red}--lwrap</td>
</tr>
<tr>
<td></td>
<td>#0f0e</td>
<td></td>
<td>[HTML]{FE01EA}--lwrap</td>
</tr>
<tr>
<td></td>
<td>rgb(128,255,0)</td>
<td></td>
<td>[HTML]{FF00EE}--lwrap</td>
</tr>
<tr>
<td></td>
<td>rgba(128,0,0,0.5)</td>
<td></td>
<td>[rgb]{0.5,1,0}--lwrap</td>
</tr>
<tr>
<td></td>
<td>rgb(25%,255,50%)</td>
<td></td>
<td>[rgb]{0.5,0,0}--lwrap</td>
</tr>
<tr>
<td></td>
<td>[rgb]{0.25,1,0.5}</td>
<td></td>
<td>[rgb]{0.25,1,0.5}--lwrap</td>
</tr>
<tr>
<td>color</td>
<td>red</td>
<td>color</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#fe01ea</td>
<td></td>
<td>{red}</td>
</tr>
<tr>
<td></td>
<td>#0f0e</td>
<td></td>
<td>[HTML]{FE01EA}</td>
</tr>
<tr>
<td></td>
<td>rgb(128,255,0)</td>
<td></td>
<td>[HTML]{FF00EE}</td>
</tr>
<tr>
<td></td>
<td>rgba(128,0,0,0.5)</td>
<td></td>
<td>[rgb]{0.5,1,0}</td>
</tr>
<tr>
<td></td>
<td>rgb(25%,255,50%)</td>
<td></td>
<td>[rgb]{0.5,0,0}</td>
</tr>
<tr>
<td></td>
<td>[rgb]{0.25,1,0.5}</td>
<td></td>
<td>[rgb]{0.25,1,0.5}</td>
</tr>
</tbody>
</table>

It is also possible to add user-defined LaTeX only styles to a HTML-CSS Styler using the --latex flag, and to add LaTeX parsing options that the converter will detect within a CSS-comment.

```python
>>> df = pd.DataFrame([[1]])
>>> df.style.set_properties(
... **"font-weight": "bold /--dwrap */", "Huge": "--latex--rwrap"**
... ).to_latex(convert_css=True)
\begin{tabular}{lr}
```

(continues on next page)
Examples

Below we give a complete step by step example adding some advanced features and noting some common gotchas.

First we create the DataFrame and Styler as usual, including MultiIndex rows and columns, which allow for more advanced formatting options:

```python
>>> cidx = pd.MultiIndex.from_arrays([...
...    ['Equity', 'Equity', 'Equity', 'Equity', ...
...    'Stats', 'Stats', 'Stats', 'Rating'],
...    ['Energy', 'Energy', 'Consumer', 'Consumer', '', '', '', ''],
...    ['BP', 'Shell', 'H&M', 'Unilever'],
...    ['Std Dev', 'Variance', '52w High', '52w Low', '']
...])
>>> iidx = pd.MultiIndex.from_arrays([...
...    ['Equity', 'Equity', 'Equity', 'Equity'],
...    ['Energy', 'Energy', 'Consumer', 'Consumer'],
...    ['BP', 'Shell', 'H&M', 'Unilever']
...])
>>> styler = pd.DataFrame([...
...    [1, 0.8, 0.66, 0.72, 32.1678, 32.1678**2, 335.12, 240.89, 'Buy'],
...    [0.8, 1.0, 0.69, 0.79, 1.876, 1.876**2, 14.12, 19.78, 'Hold'],
...    [0.66, 0.69, 1.0, 0.86, 7, 7**2, 210.9, 140.6, 'Buy'],
...    [0.72, 0.79, 0.86, 1.0, 213.76, 213.76**2, 2807, 3678, 'Sell']
...], columns=cidx, index=iidx).style
```

Second we will format the display and, since our table is quite wide, will hide the repeated level-0 of the index:

```python
>>> styler.format(subset="Equity", precision=2)
...   .format(subset="Stats", precision=1, thousands="","")
...   .format(subset="Rating", formatter=str.upper)
...   .format_index(escape="latex", axis=1)
...   .format_index(escape="latex", axis=0)
...   .hide(level=0, axis=0)
```

Note that one of the string entries of the index and column headers is “H&M”. Without applying the escape="latex" option to the format_index method the resultant LaTeX will fail to render, and the error returned is quite difficult to debug. Using the appropriate escape the “&” is converted to “\&”.

Thirdly we will apply some (CSS-HTML) styles to our object. We will use a builtin method and also define our own method to highlight the stock recommendation:

```python
>>> def rating_color(v):
...     if v == "Buy": color = "#33ff85"
...     elif v == "Sell": color = "#ff5933"
...     else: color = "#ffdd33"
...     return f"color: {color}; font-weight: bold;"
>>> styler.background_gradient(cmap="inferno", subset="Equity", vmin=0, vmax=1)
...   .applymap(rating_color, subset="Rating")
```
All the above styles will work with HTML (see below) and LaTeX upon conversion:

```
>> styler.applymap_index(
    ... lambda v: "rotatebox:45--rwrap--latex;", level=2, axis=1
    ...
)
```

Finally we render our LaTeX adding in other options as required:

```
>> styler.to_latex(
    ... caption="Selected stock correlation and simple statistics."
    ...
    ... clines="skip-last:3",
    ...
    ... convert_css=True,
    ...
    ... position_float="centering",
    ...
    ... multicol_align="|c|",
    ...
    ... hrules=True,
    ...
)
```

```latex
\begin{table}
\centering
\caption{Selected stock correlation and simple statistics.}
\begin{tabular}{llrrrrrrrrl}
\toprule
& & \multicolumn{4}{|c|}{Equity} & \multicolumn{4}{|c|}{Stats} & Rating \\
& & BP & Shell & H\&M & Unilever & Std Dev & Variance & 52w High & 52w Low & \\
\midrule
\multirow[2]{2}{*}{Energy} & BP & 1.00 & 0.80 & 0.66 & 0.72 & 32.2 & 1,034.8 & 335.1 & 240.9 & BUY \\
& Shell & 0.80 & 1.00 & 0.69 & 0.79 & 1.9 & 3.5 & 14.1 & 19.8 & HOLD \\
\multirow{2}{*}{Consumer} & H\&M & 0.66 & 0.69 & 1.00 & 0.86 & 7.0 & 49.0 & 210.9 & 140.6 & BUY \\
& Unilever & 0.72 & 0.79 & 0.86 & 1.00 & 213.8 & 45,693.3 & 2,807.0 & 3,678.0 & SELL \\
\bottomrule
\end{tabular}
\end{table}
```
3.1.9 HDFStore: PyTables (HDF5)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_hdf</code></td>
<td>Read from the store, close it if we opened it.</td>
</tr>
<tr>
<td><code>HDFStore.put</code></td>
<td>Store object in HDFStore.</td>
</tr>
<tr>
<td><code>HDFStore.append</code></td>
<td>Append to Table in file.</td>
</tr>
<tr>
<td><code>HDFStore.get</code></td>
<td>Retrieve pandas object stored in file.</td>
</tr>
<tr>
<td><code>HDFStore.select</code></td>
<td>Retrieve pandas object stored in file, optionally based on where criteria.</td>
</tr>
<tr>
<td><code>HDFStore.info</code></td>
<td>Print detailed information on the store.</td>
</tr>
<tr>
<td><code>HDFStore.keys</code></td>
<td>Return a list of keys corresponding to objects stored in HDFStore.</td>
</tr>
<tr>
<td><code>HDFStore.groups</code></td>
<td>Return a list of all the top-level nodes.</td>
</tr>
<tr>
<td><code>HDFStore.walk</code></td>
<td>Walk the pytables group hierarchy for pandas objects.</td>
</tr>
</tbody>
</table>
pandas.read_hdf

```
pandas.read_hdf(path_or_buf, key=None, mode='r', errors='strict', where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, **kwargs)
```

Read from the store, close it if we opened it.

Retrieve pandas object stored in file, optionally based on where criteria.

**Warning:** Pandas uses PyTables for reading and writing HDF5 files, which allows serializing object-dtype data with pickle when using the “fixed” format. Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3/library/pickle.html for more.

**Parameters**

- **path_or_buf** [str, path object, pandas.HDFStore] Any valid string path is acceptable. Only supports the local file system, remote URLs and file-like objects are not supported.
  
  If you want to pass in a path object, pandas accepts any `os.PathLike`.
  
  Alternatively, pandas accepts an open `pandas.HDFStore` object.

- **key** [object, optional] The group identifier in the store. Can be omitted if the HDF file contains a single pandas object.

- **mode** [{‘r’, ‘r+’, ‘a’}, default ‘r‘] Mode to use when opening the file. Ignored if path_or_buf is a `pandas.HDFStore`. Default is ‘r’.

- **errors** [str, default ‘strict’] Specifies how encoding and decoding errors are to be handled. See the errors argument for open() for a full list of options.

- **where** [list, optional] A list of Term (or convertible) objects.

- **start** [int, optional] Row number to start selection.

- **stop** [int, optional] Row number to stop selection.

- **columns** [list, optional] A list of columns names to return.

- **iterator** [bool, optional] Return an iterator object.

- **chunksize** [int, optional] Number of rows to include in an iteration when using an iterator.

- ****kwargs Additional keyword arguments passed to HDFStore.

**Returns**

- **item** [object] The selected object. Return type depends on the object stored.

See also:

- **DataFrame.to_hdf** Write a HDF file from a DataFrame.

- **HDFStore** Low-level access to HDF files.
Examples

```python
>>> df = pd.DataFrame([[1, 1.0, 'a']], columns=['x', 'y', 'z'])
>>> df.to_hdf('~/store.h5', 'data')
>>> reread = pd.read_hdf('~/store.h5')
```

pandas.HDFStore.put

HDFStore.put(key, value, format=None, index=True, append=False, complib=None, complevel=None,
min_itemsize=None, nan_rep=None, data_columns=None, encoding=None, errors='strict',
track_times=True, dropna=False)

Store object in HDFStore.

Parameters

- **key** [str]
- **value** [{Series, DataFrame}]
- **format** ['fixed(f)|table(t)', default is 'fixed'] Format to use when storing object in HDFStore. Value can be one of:
  - 'table' Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- **append** [bool, default False] This will force Table format, append the input data to the existing.
- **data_columns** [list of columns or True, default None] List of columns to create as data columns, or True to use all columns. See here.
- **encoding** [str, default None] Provide an encoding for strings.
- **track_times** [bool, default True] Parameter is propagated to ‘create_table’ method of ‘PyTables’. If set to False it enables to have the same h5 files (same hashes) independent on creation time.

New in version 1.1.0.

pandas.HDFStore.append

HDFStore.append(key, value, format=None, axes=None, index=True, append=True, complib=None, complevel=None, columns=None, min_itemsize=None, nan_rep=None, chunksize=None, expectedrows=None, dropna=None, data_columns=None, encoding=None, errors='strict')

Append to Table in file. Node must already exist and be Table format.

Parameters

- **key** [str]
- **value** [{Series, DataFrame}]
- **format** ['table' is the default] Format to use when storing object in HDFStore. Value can be one of:
  - 'table' Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- **append** [bool, default True] Append the input data to the existing.
**data_columns** [list of columns, or True, default None] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See here.

**min_itemsize** [dict of columns that specify minimum str sizes]

**nan_rep** [str to use as str nan representation]

**chunksize** [size to chunk the writing]

**expectedrows** [expected TOTAL row size of this table]

**encoding** [default None, provide an encoding for str]

**dropna** [bool, default False] Do not write an ALL nan row to the store settable by the option ‘io.hdf.dropna_table’.

**Notes**

Does not check if data being appended overlaps with existing data in the table, so be careful

### pandas.HDFStore.get

**HDFStore.get(key)**

Retrieve pandas object stored in file.

**Parameters**

- **key** [str]

**Returns**

- **object** Same type as object stored in file.

### pandas.HDFStore.select

**HDFStore.select(key, where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, auto_close=False)**

Retrieve pandas object stored in file, optionally based on where criteria.

**Warning:** Pandas uses PyTables for reading and writing HDF5 files, which allows serializing object-dtype data with pickle when using the “fixed” format. Loading pickled data received from untrusted sources can be unsafe.

See: [https://docs.python.org/3/library/pickle.html](https://docs.python.org/3/library/pickle.html) for more.

**Parameters**

- **key** [str] Object being retrieved from file.
- **where** [list or None] List of Term (or convertible) objects, optional.
- **start** [int or None] Row number to start selection.
- **stop** [int, default None] Row number to stop selection.
- **columns** [list or None] A list of columns that if not None, will limit the return columns.
iterator  [bool or False] Returns an iterator.
chunksize  [int or None] Number or rows to include in iteration, return an iterator.
auto_close  [bool or False] Should automatically close the store when finished.

Returns

object  Retrieved object from file.

pandas.HDFStore.info

HDFStore.info()
Print detailed information on the store.

Returns

str

pandas.HDFStore.keys

HDFStore.keys(include='pandas')
Return a list of keys corresponding to objects stored in HDFStore.

Parameters

include  [str, default 'pandas'] When kind equals 'pandas' return pandas objects. When kind equals 'native' return native HDF5 Table objects.

New in version 1.1.0.

Returns

list  List of ABSOLUTE path-names (e.g. have the leading '/').

Raises

raises ValueError if kind has an illegal value

pandas.HDFStore.groups

HDFStore.groups()
Return a list of all the top-level nodes.
Each node returned is not a pandas storage object.

Returns

list  List of objects.
pandas.HDFStore.walk

HDFStore.walk(where='/')
Walk the pytables group hierarchy for pandas objects.
This generator will yield the group path, subgroups and pandas object names for each group.
Any non-pandas PyTables objects that are not a group will be ignored.
The where group itself is listed first (preorder), then each of its child groups (following an alphanumerical order) is also traversed, following the same procedure.

Parameters

where [str, default ‘/’] Group where to start walking.

Yields

path [str] Full path to a group (without trailing ‘/’).
groups [list] Names (strings) of the groups contained in path.
leaves [list] Names (strings) of the pandas objects contained in path.

Warning: One can store a subclass of DataFrame or Series to HDF5, but the type of the subclass is lost upon storing.

3.1.10 Feather

read_feather(path[, columns, use_threads, ...]) Load a feather-format object from the file path.
DataFrame.to_feather(path, **kwargs) Write a DataFrame to the binary Feather format.

pandas.read_feather

pandas.read_feather(path, columns=None, use_threads=True, storage_options=None)
Load a feather-format object from the file path.

Parameters

path [str, path object, or file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a binary read() function. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file:///localhost/path/to/table.feather.
columns [sequence, default None] If not provided, all columns are read.
use_threads [bool, default True] Whether to parallelize reading using multiple threads.
storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

Returns

New in version 1.2.0.
**pandas.DataFrame.to_feather**

`DataFrame.to_feather(path, **kwargs)`

Write a DataFrame to the binary Feather format.

**Parameters**

- `path` [str, path object, file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a binary write() function. If a string or a path, it will be used as Root Directory path when writing a partitioned dataset.

- `**kwargs` : Additional keywords passed to `pyarrow.feather.write_feather()`. Starting with pyarrow 0.17, this includes the `compression`, `compression_level`, `chunksize` and `version` keywords.

New in version 1.1.0.

**Notes**

This function writes the dataframe as a feather file. Requires a default index. For saving the DataFrame with your custom index use a method that supports custom indices e.g. `to_parquet`.

### 3.11 Parquet

<table>
<thead>
<tr>
<th><code>read_parquet(path[, engine, columns, ...])</code></th>
<th>Load a parquet object from the file path, returning a DataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.to_parquet([path, engine, ...])</code></td>
<td>Write a DataFrame to the binary parquet format.</td>
</tr>
</tbody>
</table>

**pandas.read_parquet**

`pandas.read_parquet(path, engine='auto', columns=None, storage_options=None, use_nullable_dtypes=False, **kwargs)`

Load a parquet object from the file path, returning a DataFrame.

**Parameters**

- `path` [str, path object or file-like object] String, path object (implementing os.PathLike[str]), or file-like object implementing a binary read() function. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.parquet`. A file URL can also be a path to a directory that contains multiple partitioned parquet files. Both pyarrow and fastparquet support paths to directories as well as file URLs. A directory path could be: `file://localhost/path/to/tables` or `s3://bucket/partition_dir`.

- `engine` [[‘auto’, ‘pyarrow’, ‘fastparquet’], default ‘auto’] Parquet library to use. If ‘auto’, then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try ‘pyarrow’, falling back to ‘fastparquet’ if ‘pyarrow’ is unavailable.

- `columns` [list, default=None] If not None, only these columns will be read from the file.
storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.3.0.

use_nullable_dtypes [bool, default False] If True, use dtypes that use pd.NA as missing value indicator for the resulting DataFrame. (only applicable for the pyarrow engine) As new dtypes are added that support pd.NA in the future, the output with this option will change to use those dtypes. Note: this is an experimental option, and behaviour (e.g. additional support dtypes) may change without notice.

New in version 1.2.0.

**kwargs Any additional kwargs are passed to the engine.

Returns
DataFrame

pandas.DataFrame.to_parquet

DataFrame.to_parquet(path=None, engine='auto', compression='snappy', index=None, partition_cols=None, storage_options=None, **kwargs)

Write a DataFrame to the binary parquet format.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

Parameters

path [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[str]), or file-like object implementing a binary write() function. If None, the result is returned as bytes. If a string or path, it will be used as Root Directory path when writing a partitioned dataset.

Changed in version 1.2.0.

Previously this was “fname”

engine [{‘auto’, ‘pyarrow’, ‘fastparquet’}, default ‘auto’] Parquet library to use. If ‘auto’, then the option io.parquet.engine is used. The default io.parquet.engine behavior is to try ‘pyarrow’, falling back to ‘fastparquet’ if ‘pyarrow’ is unavailable.

compression [{‘snappy’, ‘gzip’, ‘brotli’}, None, default ‘snappy’] Name of the compression to use. Use None for no compression.

index [bool, default None] If True, include the dataframe’s index(es) in the file output. If False, they will not be written to the file. If None, similar to True the dataframe’s index(es) will be saved. However, instead of being saved as values, the RangeIndex will be stored as a range in the metadata so it doesn’t require much space and is faster. Other indexes will be included as columns in the file output.

partition_cols [list, optional, default None] Column names by which to partition the dataset. Columns are partitioned in the order they are given. Must be None if path is not a string.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and
"gcs://") the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

**kwargs Additional arguments passed to the parquet library. See pandas io for more details.

Returns

bytes if no path argument is provided else None

See also:

read_parquet Read a parquet file.
Dataframe.to_csv Write a csv file.
Dataframe.to_sql Write to a sql table.
Dataframe.to_hdf Write to hdf.

Notes

This function requires either the fastparquet or pyarrow library.

Examples

```python
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip',
... compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
0    1    3
1    2    4
```

If you want to get a buffer to the parquet content you can use a io.BytesIO object, as long as you don’t use partition_cols, which creates multiple files.

```python
>>> import io
>>> f = io.BytesIO()
>>> df.to_parquet(f)
>>> f.seek(0)
>>> content = f.read()
```

3.1.12 ORC

read_orc(path[, columns]) Load an ORC object from the file path, returning a DataFrame.
**pandas.read_orc**

```python
pandas.read_orc(path, columns=None, **kwargs)
```

Load an ORC object from the file path, returning a DataFrame.

New in version 1.0.0.

**Parameters**

- `path` [str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `read()` function. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.orc`.

- `columns` [list, default None] If not None, only these columns will be read from the file.

- `**kwargs` Any additional kwargs are passed to pyarrow.

**Returns**

DataFrame

**Notes**

Before using this function you should read the *user guide about ORC* and *install optional dependencies*.

### 3.1.13 SAS

```python
read_sas(filepath_or_buffer[, format, ...])
```

Read SAS files stored as either XPORT or SAS7BDAT format files.

**pandas.read_sas**

```python
pandas.read_sas(filepath_or_buffer, format=None, index=None, encoding=None, chunksize=None, iterator=False)
```

Read SAS files stored as either XPORT or SAS7BDAT format files.

**Parameters**

- `filepath_or_buffer` [str, path object, or file-like object] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `read()` function. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.sas`.

- `format` [str {'xport', 'sas7bdat'} or None] If None, file format is inferred from file extension. If ‘xport’ or ‘sas7bdat’, uses the corresponding format.

- `index` [identifier of index column, defaults to None] Identifier of column that should be used as index of the DataFrame.

- `encoding` [str, default is None] Encoding for text data. If None, text data are stored as raw bytes.

- `chunksize` [int] Read file `chunksize` lines at a time, returns iterator.

Changed in version 1.2: `TextFileReader` is a context manager.
iterator  [bool, defaults to False] If True, returns an iterator for reading the file incrementally.

Changed in version 1.2: TextFileReader is a context manager.

Returns

DataFrame if iterator=False and chunksize=None, else SAS7BDATReader
or XportReader

3.1.14 SPSS

read_spss(path[, usecols, convert_categoricals])  Load an SPSS file from the file path, returning a DataFrame.

pandas.read_spss

pandas.read_spss(path, usecols=None, convert_categoricals=True)

Load an SPSS file from the file path, returning a DataFrame.

New in version 0.25.0.

Parameters

path  [str or Path] File path.

usecols  [list-like, optional] Return a subset of the columns. If None, return all columns.

convert_categoricals  [bool, default is True] Convert categorical columns into pd.Categorical.

Returns

DataFrame

3.1.15 SQL

read_sql_table(table_name, con[, schema, ...])  Read SQL database table into a DataFrame.

read_sql_query(sql, con[, index_col, ...])  Read SQL query into a DataFrame.

read_sql(sql, con[, index_col, ...])  Read SQL query or database table into a DataFrame.

DataFrame.to_sql(name, con[, schema, ...])  Write records stored in a DataFrame to a SQL database.

pandas.read_sql_table

pandas.read_sql_table(table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None)

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters

table_name  [str] Name of SQL table in database.

con  [SQLAlchemy connectable or str] A database URI could be provided as str. SQLite DBAPI connection mode not supported.
schema [str, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col [str or list of str, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates [list or dict, default None]
  - List of column names to parse as dates.
  - Dict of {column_name: format string} where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
  - Dict of {column_name: arg dict}, where the arg dict corresponds to the keyword arguments of pandas.to_datetime() Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default None] List of column names to select from SQL table.

chunksize [int, default None] If specified, returns an iterator where chunksize is the number of rows to include in each chunk.

Returns

DataFrame or Iterator[DataFrame] A SQL table is returned as two-dimensional data structure with labeled axes.

See also:

read_sql_query Read SQL query into a DataFrame.
read_sql Read SQL query or database table into a DataFrame.

Notes

Any datetime values with time zone information will be converted to UTC.

Examples

```python
gpd.read_sql_table('table_name', 'postgres:///db_name')
```

pandas.read_sql_query

pandas.read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=None, dtype=None)

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an index_col parameter to use one of the columns as the index, otherwise default integer index will be used.

Parameters

sql [str SQL query or SQLAlchemy Selectable (select or text object)] SQL query to be executed.
**pandas**: powerful Python data analysis toolkit, Release 1.4.2

**con** [SQLAlchemy connectable, str, or sqlite3 connection] Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

**index_col** [str or list of str, optional, default: None] Column(s) to set as index (MultiIndex).

**coerce_float** [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Useful for SQL result sets.

**params** [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported. E.g. for psycopg2, uses %s(name)s so use params={'name': 'value'}.

**parse_dates** [list or dict, default: None]
- List of column names to parse as dates.
- Dict of {column_name: format string} where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of {column_name: arg dict}, where the arg dict corresponds to the keyword arguments of pandas.to_datetime() Especially useful with databases without native Datetime support, such as SQLite.

**chunksize** [int, default None] If specified, return an iterator where chunksize is the number of rows to include in each chunk.

**dtype** [Type name or dict of columns] Data type for data or columns. E.g. np.float64 or {'a': np.float64, 'b': np.int32, 'c': 'Int64'}.

New in version 1.3.0.

Returns

DataFrame or Iterator[DataFrame]

See also:

read_sql_table Read SQL database table into a DataFrame.

read_sql Read SQL query or database table into a DataFrame.

Notes

Any datetime values with time zone information parsed via the parse_dates parameter will be converted to UTC.

**pandas.read_sql**

pandas.read_sql(sql, con=None, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None)

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around read_sql_table and read_sql_query (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to read_sql_query, while a database table name will be routed to read_sql_table. Note that the delegated function might have more specific notes about their functionality not listed here.
Parameters

-sql [str or SQLAlchemy Selectable (select or text object)] SQL query to be executed or a table name.

-con [SQLAlchemy connectable, str, or sqlite3 connection] Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable; str connections are closed automatically. See here.

-index_col [str or list of str, optional, default: None] Column(s) to set as index(MultiIndex).

-coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

-params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : ‘value’}.

-parse_dates [list or dict, default: None]
  • List of column names to parse as dates.
  • Dict of {column_name: format string} where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
  • Dict of {column_name: arg dict}, where the arg dict corresponds to the keyword arguments of pandas.to_datetime() Especially useful with databases without native Datetime support, such as SQLite.

-columns [list, default: None] List of column names to select from SQL table (only used when reading a table).

-chunksize [int, default None] If specified, return an iterator where chunksize is the number of rows to include in each chunk.

Returns

DataFrame or Iterator[DataFrame]

See also:

read_sql_table Read SQL database table into a DataFrame.
read_sql_query Read SQL query into a DataFrame.

Examples

Read data from SQL via either a SQL query or a SQL tablename. When using a SQLite database only SQL queries are accepted, providing only the SQL tablename will result in an error.

```python
>>> from sqlite3 import connect
cpp: powerful Python data analysis toolkit, Release 1.4.2
>>> conn = connect(':memory:)
>>> df = pd.DataFrame(data=[[0, '10/11/12'], [1, '12/11/10']],
... columns=['int_column', 'date_column'])
>>> df.to_sql('test_data', conn)
```
Apply date parsing to columns through the `parse_dates` argument

```python
>>> pd.read_sql('SELECT int_column, date_column FROM test_data', conn,
               ...                              parse_dates=['date_column'])
int_column  date_column
0           0  2012-10-11
1           1  2010-12-11
```

The `parse_dates` argument calls `pd.to_datetime` on the provided columns. Custom argument values for applying `pd.to_datetime` on a column are specified via a dictionary format: 1. Ignore errors while parsing the values of “date_column”

```python
>>> pd.read_sql('SELECT int_column, date_column FROM test_data',
               ... conn,
               ...                  parse_dates={"date_column": {"errors": "ignore"}})
int_column  date_column
0           0  2012-10-11
1           1  2010-12-11
```

2. Apply a dayfirst date parsing order on the values of “date_column”

```python
>>> pd.read_sql('SELECT int_column, date_column FROM test_data',
               ... conn,
               ...                  parse_dates={"date_column": {"dayfirst": True}})
int_column  date_column
0           0  2012-11-10
1           1  2010-11-12
```

3. Apply custom formatting when date parsing the values of “date_column”

```python
>>> pd.read_sql('SELECT int_column, date_column FROM test_data',
               ... conn,
               ...                  parse_dates={"date_column": {"format": "%d/%m/%y"}})
int_column  date_column
0           0  2012-11-10
1           1  2010-11-12
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DataFrame.to_sql**

`DataFrame.to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None)`

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

**Parameters**

- **name** [str] Name of SQL table.
- **con** [sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable See here.
- **schema** [str, optional] Specify the schema (if database flavor supports this). If None, use default schema.
- **if_exists** ['fail', 'replace', 'append'], default 'fail' How to behave if the table already exists.
  - fail: Raise a ValueError.
  - replace: Drop the table before inserting new values.
  - append: Insert new values to the existing table.
- **index** [bool, default True] Write DataFrame index as a column. Uses `index_label` as the column name in the table.
- **index_label** [str or sequence, default None] Column label for index column(s). If None is given (default) and `index` is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- **chunksize** [int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.
- **dtype** [dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.
- **method** [{None, ‘multi’, callable}, optional] Controls the SQL insertion clause used:
  - None : Uses standard SQL INSERT clause (one per row).
  - ‘multi’: Pass multiple values in a single INSERT clause.
  - callable with signature (pd_table, conn, keys, data_iter).
Details and a sample callable implementation can be found in the section `insert method`.

**Returns**

- **None or int** Number of rows affected by to_sql. None is returned if the callable passed into method does not return the number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of sqlite3. Cursor or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.

New in version 1.4.0.

**Raises**
**ValueError** When the table already exists and if\_exists is ‘fail’ (the default).

See also:

`read_sql` Read a DataFrame from a table.

**Notes**

Timezone aware datetime columns will be written as Timestamp with timezone type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

**References**

[1], [2]

**Examples**

Create an in-memory SQLite database.

```python
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```python
>>> df = pd.DataFrame({'name': ['User 1', 'User 2', 'User 3']})
>>> df
     name
0   User 1
1    User 2
2    User 3
```

```python
>>> df.to_sql('users', con=engine)
3
```

```python
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An `sqlalchemy.engine.Connection` can also be passed to `con`:

```python
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name': ['User 4', 'User 5']})
...     df1.to_sql('users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```python
>>> df2 = pd.DataFrame({'name': ['User 6', 'User 7']})
>>> df2.to_sql('users', con=engine, if_exists='append')
2
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```
Overwrite the table with just df2.

```python
>>> df2.to_sql('users', con=engine, if_exists='replace',
...    index_label='id')
2
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```python
>>> df = pd.DataFrame({'A': [1, None, 2]})
>>> df
   A
0  1.0
1  NaN
2  2.0

>>> from sqlalchemy.types import Integer

>>> df.to_sql('integers', con=engine, index=False,
...    dtype={'A': Integer()})
3

>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

### 3.1.16 Google BigQuery

`read_gbq(query[, project_id, index_col, ...])` Load data from Google BigQuery.

#### pandas.read_gbq

`pandas.read_gbq(query, project_id=None, index_col=None, col_order=None, reauth=False, auth_local_webserver=False, dialect=None, location=None, configuration=None, credentials=None, use_bqstorage_api=None, max_results=None, progress_bar_type=None)`

Load data from Google BigQuery.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

**Parameters**

- `query` [str] SQL-Like Query to return data values.
- `project_id` [str, optional] Google BigQuery Account project ID. Optional when available from the environment.
- `index_col` [str, optional] Name of result column to use for index in results DataFrame.
- `col_order` [list(str), optional] List of BigQuery column names in the desired order for results DataFrame.
reauth [bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

    New in version 0.2.0 of pandas-gbq.

dialect [str, default ‘legacy’] Note: The default value is changing to ‘standard’ in a future version.

    SQL syntax dialect to use. Value can be one of:

    ‘legacy’ Use BigQuery’s legacy SQL dialect. For more information see BigQuery Legacy SQL Reference.

    ‘standard’ Use BigQuery’s standard SQL, which is compliant with the SQL 2011 standard. For more information see BigQuery Standard SQL Reference.

location [str, optional] Location where the query job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of any datasets used in the query.

    New in version 0.5.0 of pandas-gbq.

configuration [dict, optional] Query config parameters for job processing. For example:

    configuration = {'query': {'useQueryCache': False}}

    For more information see BigQuery REST API Reference.

credentials [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine google.auth.compute_engine.Credentials or Service Account google.oauth2.service_account.Credentials directly.

    New in version 0.8.0 of pandas-gbq.

use_bqstorage_api [bool, default False] Use the BigQuery Storage API to download query results quickly, but at an increased cost. To use this API, first enable it in the Cloud Console. You must also have the bigquery.readsessions.create permission on the project you are billing queries to.

    This feature requires version 0.10.0 or later of the pandas-gbq package. It also requires the google-cloud-bigquery-storage and fastavro packages.

    New in version 0.25.0.

max_results [int, optional] If set, limit the maximum number of rows to fetch from the query results.

    New in version 0.12.0 of pandas-gbq.

    New in version 1.1.0.

progress_bar_type [Optional, str] If set, use the tqdm library to display a progress bar while the data downloads. Install the tqdm package to use this feature.

    Possible values of progress_bar_type include:

    None No progress bar.

    ‘tqdm’ Use the tqdm.tqdm() function to print a progress bar to sys.stderr.

    ‘tqdm_notebook’ Use the tqdm.tqdm_notebook() function to display a progress bar as a Jupyter notebook widget.
'tqdm_gui' Use the `tqdm.tqdm_gui()` function to display a progress bar as a graphical dialog box.

Note that this feature requires version 0.12.0 or later of the pandas-gbq package. And it requires the tqdm package. Slightly different than pandas-gbq, here the default is `None`.

New in version 1.0.0.

**Returns**

**df**: DataFrame DataFrame representing results of query.

See also:

- `pandas_gbq.read_gbq` This function in the pandas-gbq library.
- `DataFrame.to_gbq` Write a DataFrame to Google BigQuery.

### 3.1.17 STATA

<table>
<thead>
<tr>
<th>read_stata(filepath_or_buffer[...,])</th>
<th>Read Stata file into DataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.to_stata(path[, convert_dates, ...])</td>
<td>Export DataFrame object to Stata dta format.</td>
</tr>
</tbody>
</table>

#### pandas.read_stata

```python
pandas.read_stata(filepath_or_buffer, convert_dates=True, convert_categoricals=True, index_col=None,
                  convert_missing=False, preserve_dtypes=True, columns=None, order_categoricals=True,
                  chunksize=None, iterator=False, compression='infer', storage_options=None)
```

Read Stata file into DataFrame.

**Parameters**

- **filepath_or_buffer**: [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.dta`. If you want to pass in a path object, pandas accepts any `os.PathLike`. By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.
- **convert_dates**: [bool, default True] Convert date variables to DataFrame time values.
- **convert_categoricals**: [bool, default True] Read value labels and convert columns to Categorical/Factor variables.
- **index_col**: [str, optional] Column to set as index.
- **convert_missing**: [bool, default False] Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissingValue objects.
- **preserve_dtypes**: [bool, default True] Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).
- **columns**: [list or None] Columns to retain. Columns will be returned in the given order. None returns all columns.
order_categoricals [bool, default True] Flag indicating whether converted categorical data are ordered.

chunksize [int, default None] Return StataReader object for iterations, returns chunks with given number of lines.

iterator [bool, default False] Return StataReader object.

compression [str or dict, default ‘infer’] For on-the-fly decompression of on-disk data. If ‘infer’ and ‘%s’ is path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key ‘method’ set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: compression={‘method’: ‘zstd’, ‘dict_data’: my_compression_dict}.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

Returns
DataFrame or StataReader

See also:

io.stata.StataReader Low-level reader for Stata data files.
DataFrame.to_stata Export Stata data files.

Notes
Categorical variables read through an iterator may not have the same categories and dtype. This occurs when a variable stored in a DTA file is associated to an incomplete set of value labels that only label a strict subset of the values.

Examples

Creating a dummy stata for this example >>> df = pd.DataFrame(‘animal’: [‘falcon’, ‘parrot’, ‘falcon’, ‘parrot’], ‘speed’: [350, 18, 361, 15]) # doctest: +SKIP >>> df.to_stata(‘animals.dta’) # doctest: +SKIP

Read a Stata dta file:

```python
>>> df = pd.read_stata(‘animals.dta’)
```

Read a Stata dta file in 10,000 line chunks: >>> values = np.random.randint(0, 10, size=(20_000, 1), dtype=“uint8”) # doctest: +SKIP >>> df = pd.DataFrame(values, columns=[‘i’]) # doctest: +SKIP >>> df.to_stata(‘filename.dta’) # doctest: +SKIP

```python
>>> itr = pd.read_stata(‘filename.dta’, chunksize=10000)
>>> for chunk in itr:
...     # Operate on a single chunk, e.g., chunk.mean()
...     pass
```
`pandas.DataFrame.to_stata`

`DataFrame.to_stata(path, convert_dates=None, write_index=True, byteorder=None, time_stamp=None, data_label=None, variable_labels=None, version=114, convert_strl=None, compression='infer', storage_options=None, *, value_labels=None)`

Export DataFrame object to Stata dta format.

Writes the DataFrame to a Stata dataset file. “dta” files contain a Stata dataset.

**Parameters**

- `path` [str, path object, or buffer] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function.

  Changed in version 1.0.0.

  Previously this was “fname”

- `convert_dates` [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to ‘tc’. Raises `NotImplementedError` if a datetime column has timezone information.

- `write_index` [bool] Write the index to Stata dataset.

- `byteorder` [str] Can be “>”, “<”, “little”, or “big”. default is `sys.byteorder`.

- `time_stamp` [datetime] A datetime to use as file creation date. Default is the current time.

- `data_label` [str, optional] A label for the data set. Must be 80 characters or smaller.

- `variable_labels` [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

- `version` [{114, 117, 118, 119, None}, default 114] Version to use in the output dta file. Set to None to let pandas decide between 118 or 119 formats depending on the number of columns in the frame. Version 114 can be read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 118 is supported in Stata 14 and later. Version 119 is supported in Stata 15 and later. Version 114 limits string variables to 244 characters or fewer while versions 117 and later allow strings with lengths up to 2,000,000 characters. Versions 118 and 119 support Unicode characters, and version 119 supports more than 32,767 variables.

  Version 119 should usually only be used when the number of variables exceeds the capacity of dta format 118. Exporting smaller datasets in format 119 may have unintended consequences, and, as of November 2020, Stata SE cannot read version 119 files.

  Changed in version 1.0.0: Added support for formats 118 and 119.

- `convert_strl` [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

- `compression` [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘path’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of {‘zip’, ‘gzip’, ‘bz2’, ‘zstd’} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.
New in version 1.1.0.

Changed in version 1.4.0: Zstandard support.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

**value_labels** [dict of dicts] Dictionary containing columns as keys and dictionaries of column value to labels as values. Labels for a single variable must be 32,000 characters or smaller.

New in version 1.4.0.

Raises

**NotImplementedError**
- If datetimes contain timezone information
- Column dtype is not representable in Stata

**ValueError**
- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

See also:

*read_stata* Import Stata data files.

*io.stata.StataWriter* Low-level writer for Stata data files.

*io.stata.StataWriter117* Low-level writer for version 117 files.

Examples

```python
>>> df = pd.DataFrame({
    'animal': ['falcon', 'parrot', 'falcon', 'falcon', 'parrot'],
    'speed': [350, 18, 361, 15]
})
>>> df.to_stata('animals.dta')
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StataReader.data_label</td>
<td>Return data label of Stata file.</td>
</tr>
<tr>
<td>StataReader.value_labels()</td>
<td>Return a dict, associating each variable name a dict, associating each value its corresponding label.</td>
</tr>
<tr>
<td>StataReader.variable_labels()</td>
<td>Return variable labels as a dict, associating each variable name with corresponding label.</td>
</tr>
<tr>
<td>StataWriter.write_file()</td>
<td>Export DataFrame object to Stata dta format.</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.io.stata.StataReader.data_label**

*property* StataReader.data_label

Return data label of Stata file.

**pandas.io.stata.StataReader.value_labels**

StataReader.value_labels()

Return a dict, associating each variable name a dict, associating each value its corresponding label.

**Returns**

dict

**pandas.io.stata.StataReader.variable_labels**

StataReader.variable_labels()

Return variable labels as a dict, associating each variable name with corresponding label.

**Returns**

dict

**pandas.io.stata.StataWriter.write_file**

StataWriter.write_file()

Export DataFrame object to Stata dta format.
# 3.2 General functions

## 3.2.1 Data manipulations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>melt</strong> (frame[, id_vars, value_vars, var_name, ...])</td>
<td>Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.</td>
</tr>
<tr>
<td><strong>pivot</strong> (data[, index, columns, values])</td>
<td>Return reshaped DataFrame organized by given index / column values.</td>
</tr>
<tr>
<td><strong>pivot_table</strong> (data[, values, index, columns, ...])</td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
</tr>
<tr>
<td><strong>crosstab</strong> (index, columns[, values, rownames, ...])</td>
<td>Compute a simple cross tabulation of two (or more) factors.</td>
</tr>
<tr>
<td><strong>cut</strong> (x, bins[, right, labels, retbins, ...])</td>
<td>Bin values into discrete intervals.</td>
</tr>
<tr>
<td><strong>qcut</strong> (x, q[, labels, retbins, precision, ...])</td>
<td>Quantile-based discretization function.</td>
</tr>
<tr>
<td><strong>merge</strong> (left, right[, how, on, left_on, ...])</td>
<td>Merge DataFrame or named Series objects with a database-style join.</td>
</tr>
<tr>
<td><strong>merge_ordered</strong> (left, right[, on, left_on, ...])</td>
<td>Perform a merge for ordered data with optional filling/interpolation.</td>
</tr>
<tr>
<td><strong>merge_asof</strong> (left, right[, on, left_on, ...])</td>
<td>Perform a merge by key distance.</td>
</tr>
<tr>
<td><strong>concat</strong> (objs[, axis, join, ignore_index, ...])</td>
<td>Concatenate pandas objects along a particular axis with optional set logic along the other axes.</td>
</tr>
<tr>
<td><strong>get_dummies</strong> (data[, prefix, prefix_sep, ...])</td>
<td>Convert categorical variable into dummy/indicator variables.</td>
</tr>
<tr>
<td><strong>factorize</strong> (values[, sort, na_sentinel, size_hint])</td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><strong>unique</strong> (values)</td>
<td>Return unique values based on a hash table.</td>
</tr>
<tr>
<td><strong>wide_to_long</strong> (df, stubnames, i, j[, sep, suffix])</td>
<td>Unpivot a DataFrame from wide to long format.</td>
</tr>
</tbody>
</table>

### pandas.melt

**pandas.melt** (frame, id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None, ignore_index=True)

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (id_vars), while all other columns, considered measured variables (value_vars), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

**Parameters**

- **id_vars** [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.
- **value_vars** [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
- **var_name** [scalar] Name to use for the ‘variable’ column. If None it uses frame.columns.name or ‘variable’.
- **value_name** [scalar, default ‘value’] Name to use for the ‘value’ column.
- **col_level** [int or str, optional] If columns are a MultiIndex then use this level to melt.
- **ignore_index** [bool, default True] If True, original index is ignored. If False, the original index is retained. Index labels will be repeated as necessary.

New in version 1.1.0.
Returns

DataFrame Unpivoted DataFrame.

See also:

DataFrame.melt Identical method.
pivot_table Create a spreadsheet-style pivot table as a DataFrame.
DataFrame.pivot Return reshaped DataFrame organized by given index / column values.
DataFrame.explode Explode a DataFrame from list-like columns to long format.

Notes

Reference the user guide for more examples.

Examples

```python
>>> df = pd.DataFrame({'A': [0: 'a', 1: 'b', 2: 'c'],
...                    'B': [0: 1, 1: 3, 2: 5],
...                    'C': [0: 2, 1: 4, 2: 6]})
```

```plaintext
A B C
0 a 1 2
1 b 3 4
2 c 5 6
```

```python
>>> pd.melt(df, id_vars=['A'], value_vars=['B'])
```

```plaintext
A variable value
0 a B 1
1 b B 3
2 c B 5
```

```python
>>> pd.melt(df, id_vars=['A'], value_vars=['B', 'C'])
```

```plaintext
A variable value
0 a B 1
1 b B 3
2 c B 5
3 a C 2
4 b C 4
5 c C 6
```

The names of ‘variable’ and ‘value’ columns can be customized:

```python
>>> pd.melt(df, id_vars=['A'], value_vars=['B'],
...          var_name='myVarname', value_name='myValname')
```

```plaintext
A myVarname myValname
0 a B 1
1 b B 3
2 c B 5
```

Original index values can be kept around:


```python
>>> pd.melt(df, id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
A variable  value
0  a     B  1
1  b     B  3
2  c     B  5
0  a     C  2
1  b     C  4
2  c     C  6
```

If you have multi-index columns:

```python
>>> df.columns = [list('ABC'), list('DEF')]

>>> df
A  B  C
D  E  F
0  a  1  2
1  b  3  4
2  c  5  6

>>> pd.melt(df, col_level=0, id_vars=['A'], value_vars=['B'])
A  variable  value
0  a     B  1
1  b     B  3
2  c     B  5

>>> pd.melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
(A, D) variable_0  variable_1  value
0  a     B        E        1
1  b     B        E        3
2  c     B        E        5
```

### pandas.pivot

**pandas.pivot**(data, index=None, columns=None, values=None)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a “pivot” table) based on column values. Uses unique values from specified *index / columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

**Parameters**

- **data** [DataFrame]
- **index** [str or object or a list of str, optional] Column to use to make new frame’s index. If None, uses existing index.
  
  Changed in version 1.1.0: Also accept list of index names.

- **columns** [str or object or a list of str] Column to use to make new frame’s columns.
  
  Changed in version 1.1.0: Also accept list of columns names.

- **values** [str, object or a list of the previous, optional] Column(s) to use for populating new frame’s values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

---

3.2. General functions

1117
Returns

**DataFrame** Returns reshaped DataFrame.

**Raises**

**ValueError:** When there are any *index*, *columns* combinations with multiple values.

*DataFrame.pivot_table* when you need to aggregate.

**See also:**

*DataFrame.pivot_table* Generalization of pivot that can handle duplicate values for one index/column pair.

*DataFrame.unstack* Pivot based on the index values instead of a column.

*wide_to_long* Wide panel to long format. Less flexible but more user-friendly than melt.

**Notes**

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Reference the user guide for more examples.

**Examples**

```python
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                         'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

>>> df
   foo  bar  baz  zoo
0   one  A    1    x
1   one  B    2    y
2   one  C    3    z
3   two  A    4    q
4   two  B    5    w
5   two  C    6    t

>>> df.pivot(index='foo', columns='bar', values='baz')
   bar
foo  baz
  one 1 2 3
  two 4 5 6

>>> df.pivot(index='foo', columns='bar')[['baz']]
   bar
foo  baz
  one 1 2 3
  two 4 5 6

>>> df.pivot(index='foo', columns='bar', values=[['baz', 'zoo']])
   baz  zoo
   bar  A  B  C  A  B  C
```

(continues on next page)
You could also assign a list of column names or a list of index names.

```python
>>> df = pd.DataFrame({
...     "lev1": [1, 1, 2, 2, 2, 2],
...     "lev2": [1, 1, 2, 1, 1, 2],
...     "lev3": [1, 2, 1, 2, 1, 2],
...     "lev4": [1, 2, 3, 4, 5, 6],
...     "values": [0, 1, 2, 3, 4, 5]
... })

>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
   lev2  lev3
lev1
   1   0.0  1.0  2.0  NaN
   2   4.0  3.0  NaN  5.0

>>> df.pivot(index="lev1", columns=["lev2"], values="values")
   lev3
lev1
   1   0.0  1.0
   2   2.0  NaN
   3   4.0  3.0
   4   NaN  5.0

A ValueError is raised if there are any duplicates.

```python
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...     "bar": ['A', 'A', 'B', 'C'],
...     "baz": [1, 2, 3, 4]})

>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
```

Notice that the first two rows are the same for our index and columns arguments.
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

ValueError: Index contains duplicate entries, cannot reshape

pandas.pivot_table

pandas.pivot_table(data=None, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All', observed=False, sort=True)

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

data [DataFrame]

values [column to aggregate, optional]

index [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions.

fill_value [scalar, default None] Value to replace missing values with (in the resulting pivot table, after aggregation).

margins [bool, default False] Add all row / columns (e.g. for subtotal / grand totals).

dropna [bool, default True] Do not include columns whose entries are all NaN.

margins_name [str, default ‘All’] Name of the row / column that will contain the totals when margins is True.

observed [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Changed in version 0.25.0.

sort [bool, default True] Specifies if the result should be sorted.

New in version 1.3.0.

Returns

DataFrame An Excel style pivot table.

See also:

DataFrame.pivot Pivot without aggregation that can handle non-numeric data.
DataFrame.melt  Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

wide_to_long  Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

Reference the user guide for more examples.

Examples

```python
... "B": ["one", "one", "one", "two", "two", "one", "one", "two", "two"],
... "C": ["small", "large", "large", "small", "small", "large", "large", "small", "small"],
... "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
... "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
```

This first example aggregates values by taking the sum.

```python
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
... columns=['C'], aggfunc=np.sum)
```

We can also fill missing values using the fill_value parameter.

```python
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
... columns=['C'], aggfunc=np.sum, fill_value=0)
```
The next example aggregates by taking the mean across multiple columns.

```python
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
                          aggfunc={'D': np.mean, 'E': np.mean})
>>> table
     D     E
A C
bar large 5.500000 7.500000
  small  5.500000  8.500000
foo large 2.000000 4.500000
  small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```python
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
                          aggfunc={'D': np.mean, 'E': [min, max, np.mean]})
>>> table
     D     E
      mean max mean  min
A C
bar large 5.500000  9 7.500000  6
  small  5.500000  9 8.500000  8
foo large 2.000000  5 4.500000  4
  small  2.333333  6 4.333333  2
```

**pandas.crosstab**

The function `pandas.crosstab` computes a simple cross tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed.

**Parameters**

- `index` [array-like, Series, or list of arrays/Series] Values to group by in the rows.
- `columns` [array-like, Series, or list of arrays/Series] Values to group by in the columns.
- `values` [array-like, optional] Array of values to aggregate according to the factors. Requires `aggfunc` be specified.
- `rownames` [sequence, default None] If passed, must match number of row arrays passed.
- `colnames` [sequence, default None] If passed, must match number of column arrays passed.
- `aggfunc` [function, optional] If specified, requires `values` be specified as well.
- `margins` [bool, default False] Add row/column margins (subtotals).
- `margins_name` [str, default ‘All’] Name of the row/column that will contain the totals when margins is True.
**dropna** [bool, default True] Do not include columns whose entries are all NaN.

**normalize** [bool, {'all', 'index', 'columns'}, or {0,1}, default False] Normalize by dividing all values by the sum of values.

- If passed ‘all’ or True, will normalize over all values.
- If passed ‘index’ will normalize over each row.
- If passed ‘columns’ will normalize over each column.
- If margins is True, will also normalize margin values.

**Returns**

**DataFrame** Cross tabulation of the data.

**See also:**

**DataFrame.pivot** Reshape data based on column values.

**pivot_table** Create a pivot table as a DataFrame.

**Notes**

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

Any input passed containing Categorical data will have all of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

In the event that there aren't overlapping indexes an empty DataFrame will be returned.

Reference the user guide for more examples.

**Examples**

```python
>>> a = np.array(['foo', 'foo', 'foo', 'foo', 'bar', 'bar',
                'bar', 'bar', 'foo', 'foo', 'foo'], dtype=object)
>>> b = np.array(['one', 'one', 'one', 'two', 'one', 'one',
                'one', 'two', 'two', 'two', 'one'], dtype=object)
>>> c = np.array(['dull', 'dull', 'shiny', 'dull', 'dull', 'shiny',
                'shiny', 'dull', 'shiny', 'shiny', 'shiny'], dtype=object)
>>> pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>dull</td>
<td>1 2</td>
<td></td>
</tr>
<tr>
<td>shiny</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>bar</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>
| foo   | 2 2  | 1   
```

Here 'c' and 'f' are not represented in the data and will not be shown in the output because dropna is True by default. Set dropna=False to preserve categories with no data.

```python
>>> foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
>>> bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
>>> pd.crosstab(foo, bar)

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>'b'</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```
row_0
a  1  0
b  0  1

>>> pd.crosstab(foo, bar, dropna=False)
coi_0 d e f
row_0
a  1  0  0
b  0  1  0
c  0  0  0
```

**pandas.cut**

**pandas.cut**(*x*, *bins*, *right=True*, *labels=None*, *retns=False*, *precision=3*, *include_lowest=False*, *duplicates='raise'*, *ordered=True*)

Bin values into discrete intervals.

Use *cut* when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable. For example, *cut* could convert ages to groups of age ranges. Supports binning into an equal number of bins, or a pre-specified array of bins.

**Parameters**

- **x** [array-like] The input array to be binned. Must be 1-dimensional.
- **bins** [int, sequence of scalars, or IntervalIndex] The criteria to bin by.
  - int: Defines the number of equal-width bins in the range of *x*. The range of *x* is extended by .1% on each side to include the minimum and maximum values of *x*.
  - sequence of scalars: Defines the bin edges allowing for non-uniform width. No extension of the range of *x* is done.
  - IntervalIndex: Defines the exact bins to be used. Note that IntervalIndex for *bins* must be non-overlapping.
- **right** [bool, default True] Indicates whether *bins* includes the rightmost edge or not. If *right* == True (the default), then the *bins* [1, 2, 3, 4] indicate (1,2], (2,3], (3,4]. This argument is ignored when *bins* is an IntervalIndex.
- **labels** [array or False, default None]Specifies the labels for the returned bins. Must be the same length as the resulting bins. If False, returns only integer indicators of the bins. This affects the type of the output container (see below). This argument is ignored when *bins* is an IntervalIndex. If True, raises an error. When *ordered=False*, labels must be provided.
- **retns** [bool, default False] Whether to return the bins or not. Useful when *bins* is provided as a scalar.
- **precision** [int, default 3] The precision at which to store and display the bins labels.
- **include_lowest** [bool, default False] Whether the first interval should be left-inclusive or not.
- **duplicates** [‘default’ ‘raise’, ‘drop’], optional] If bin edges are not unique, raise ValueError or drop non-unequds.
- **ordered** [bool, default True] Whether the labels are ordered or not. Applies to returned types Categorical and Series (with Categorical dtype). If True, the resulting categorical will be ordered. If False, the resulting categorical will be unordered (labels must be provided).

New in version 1.1.0.
Returns

out [Categorical, Series, or ndarray] An array-like object representing the respective bin for each value of x. The type depends on the value of labels.

- None (default): returns a Series for Series x or a Categorical for all other inputs. The values stored within are Interval dtype.
- sequence of scalars: returns a Series for Series x or a Categorical for all other inputs. The values stored within are whatever the type in the sequence is.
- False: returns an ndarray of integers.

bins [numpy.ndarray or IntervalIndex.] The computed or specified bins. Only returned when retbins=True. For scalar or sequence bins, this is an ndarray with the computed bins. If set duplicates=drop, bins will drop non-unique bin. For an IntervalIndex bins, this is equal to bins.

See also:

- qcut Discretize variable into equal-sized buckets based on rank or based on sample quantiles.
- Categorical Array type for storing data that come from a fixed set of values.
- Series One-dimensional array with axis labels (including time series).
- IntervalIndex Immutable Index implementing an ordered, sliceable set.

Notes

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Series or Categorical object. Reference the user guide for more examples.

Examples

Discretize into three equal-sized bins.

```python
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
...
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64, right]): [(0.994, 3.0] < (3.0, 5.0] ...
```

```python
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3, retbins=True)
...
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64, right]): [(0.994, 3.0] < (3.0, 5.0] ...
array([0.994, 3. , 5. , 7. ])
```

DisCOVERs the same bins, but assign them specific labels. Notice that the returned Categorical’s categories are labels and is ordered.

```python
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3, labels=['bad', 'medium', 'good'])
['bad', 'good', 'medium', 'medium', 'good', 'bad']
Categories (3, object): ['bad' < 'medium' < 'good']
```
ordered=False will result in unordered categories when labels are passed. This parameter can be used to allow non-unique labels:

```python
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3,
...       labels=['B', 'A', 'B'], ordered=False)
['B', 'B', 'A', 'B', 'B', 'B']
Categories (2, object): ['A', 'B']
```

labels=False implies you just want the bins back.

```python
>>> pd.cut([0, 1, 2], bins=4, labels=False)
array([0, 1, 2])
```

Passing a Series as an input returns a Series with categorical dtype:

```python
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...                index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, 3)
... (a 1.992, 4.667]
b (1.992, 4.667]
c (4.667, 7.333]
d (7.333, 10.0]
e (7.333, 10.0]
dtype: category
Categories (3, interval[float64, right]): [(1.992, 4.667] < (4.667, ...
```

Passing a Series as an input returns a Series with mapping value. It is used to map numerically to intervals based on bins.

```python
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...                index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, [0, 2, 4, 6, 10, 10], labels=False, retbins=True, right=False,
...        duplicates='drop')
... (a 1.0
b 2.0
c 3.0
d 3.0
e NaN
dtype: float64,
array([ 0, 2, 4, 6, 10]))
```

Use drop optional when bins is not unique

```python
>>> pd.cut(s, [0, 2, 4, 6, 10, 10], labels=False, retbins=True,
...        right=False, duplicates='drop')
... (a 1.0
b 2.0
c 3.0
d 3.0
e NaN
dtype: float64,
array([ 0, 2, 4, 6, 10]))
```
Passing an IntervalIndex for `bins` results in those categories exactly. Notice that values not covered by the IntervalIndex are set to NaN. 0 is to the left of the first bin (which is closed on the right), and 1.5 falls between two bins.

```python
>>> bins = pd.IntervalIndex.from_tuples([(0, 1), (2, 3), (4, 5)])
>>> pd.cut([0, 0.5, 1.5, 2.5, 4.5], bins)
[NaN, (0.0, 1.0], NaN, (2.0, 3.0], (4.0, 5.0]]
Categories (3, interval[int64, right]): [(0, 1] < (2, 3] < (4, 5]]
```

**pandas.qcut**

`pandas.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')`

Quantile-based discretization function.

Discretize variable into equal-sized buckets based on rank or based on sample quantiles. For example 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data point.

**Parameters**

- `x` [1d ndarray or Series]
- `q` [int or list-like of float] Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately array of quantiles, e.g. [0., .25, .5, .75, 1.] for quartiles.
- `labels` [array or False, default None] Used as labels for the resulting bins. Must be of the same length as the resulting bins. If False, return only integer indicators of the bins. If True, raises an error.
- `retbins` [bool, optional] Whether to return the (bins, labels) or not. Can be useful if bins is given as a scalar.
- `precision` [int, optional] The precision at which to store and display the bins labels.
- `duplicates` [{default ‘raise’, ‘drop’}, optional] If bin edges are not unique, raise ValueError or drop non-uniques.

**Returns**

- `out` [Categorical or Series or array of integers if labels is False] The return type (Categorical or Series) depends on the input: a Series of type category if input is a Series else Categorical. Bins are represented as categories when categorical data is returned.
- `bins` [ndarray of floats] Returned only if `retbins` is True.

**Notes**

Out of bounds values will be NA in the resulting Categorical object
Examples

```python
>>> pd.qcut(range(5), 4)
...  
...  [(-0.001, 1.0], (-0.001, 1.0], (1.0, 2.0], (2.0, 3.0], (3.0, 4.0]]
Categories (4, interval[float64, right]): [(-0.001, 1.0] < (1.0, 2.0] ...
```

```python
>>> pd.qcut(range(5), 3, labels=["good", "medium", "bad"]
...  
...  [good, good, medium, bad, bad]
Categories (3, object): [good < medium < bad]
```

```python
>>> pd.qcut(range(5), 4, labels=False)
array([0, 0, 1, 2, 3])
```

pandas.merge

```python
pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)
```

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

**Warning:** If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

**Parameters**

- `left` [DataFrame]
- `right` [DataFrame or named Series] Object to merge with.
- `how` [{‘left’, ‘right’, ‘outer’, ‘inner’, ‘cross’}, default ‘inner’] Type of merge to be performed.
  - `left`: use only keys from left frame, similar to a SQL left outer join; preserve key order.
  - `right`: use only keys from right frame, similar to a SQL right outer join; preserve key order.
  - `outer`: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
  - `inner`: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
  - `cross`: creates the cartesian product from both frames, preserves the order of the left keys.

New in version 1.2.0.
on  [label or list] Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes [list-like, default is ("_x", ",_y") A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in left and right respectively. Pass a value of None instead of a string to indicate that the column name from left or right should be left as-is, with no suffix. At least one of the values must not be None.

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, default False] If True, adds a column to the output DataFrame called “_merge” with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of “left_only” for observations whose merge key only appears in the left DataFrame, “right_only” for observations whose merge key only appears in the right DataFrame, and “both” if the observation’s merge key is found in both DataFrames.

validate [str, optional] If specified, checks if merge is of specified type.

• “one_to_one” or “1:1”: check if merge keys are unique in both left and right datasets.
• “one_to_many” or “1:m”: check if merge keys are unique in left dataset.
• “many_to_one” or “m:1”: check if merge keys are unique in right dataset.
• “many_to_many” or “m:m”: allowed, but does not result in checks.

Returns

DataFrame A DataFrame of the two merged objects.

See also:

merge_ordered Merge with optional filling/interpolation.
merge_asof Merge on nearest keys.
DataFrame.join Similar method using indices.
Notes

Support for specifying index levels as the `on`, `left_on`, and `right_on` parameters was added in version 0.23.0
Support for merging named Series objects was added in version 0.24.0

Examples

```python
def1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...         'value': [1, 2, 3, 5]})

def2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...         'value': [5, 6, 7, 8]})

def1
  lkey value
0  foo   1
1  bar   2
2  baz   3
3  foo   5

def2
  rkey value
0  foo   5
1  bar   6
2  baz   7
3  foo   8
```

Merge `df1` and `df2` on the `lkey` and `rkey` columns. The value columns have the default suffixes, `_x` and `_y`, appended.

```python
def1.merge(df2, left_on='lkey', right_on='rkey')
  lkey value_x rkey value_y
0  foo   1  foo   5
1  foo   1  foo   8
2  foo   5  foo   5
3  foo   5  foo   8
4  bar   2  bar   6
5  baz   3  baz   7
```

Merge DataFrames `df1` and `df2` with specified left and right suffixes appended to any overlapping columns.

```python
def1.merge(df2, left_on='lkey', right_on='rkey',
...         suffixes=('l_left', 'r_right'))
  lkey value_l_left rkey value_right
0  foo   1  foo      5
1  foo   1  foo      8
2  foo   5  foo      5
3  foo   5  foo      8
4  bar   2  bar      6
5  baz   3  baz      7
```

Merge DataFrames `df1` and `df2`, but raise an exception if the DataFrames have any overlapping columns.

```python
def1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
  ...
ValueError: columns overlap but no suffix specified:
Index(['value'], dtype='object')

```python
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
```

```python
>>> df1
   a b
0 foo 1
1 bar 2
```

```python
>>> df2
    a c
0 foo 3
1 baz 4
```

```python
>>> df1.merge(df2, how='inner', on='a')
    a b  c
0 foo 1 3
```

```python
>>> df1.merge(df2, how='left', on='a')
    a b  c
0 foo 1 3.0
1 bar 2 NaN
```

```python
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
```

```python
>>> df1
   left
0  foo
1  bar
```

```python
>>> df2
   right
0   7
1   8
```

```python
>>> df1.merge(df2, how='cross')
   left  right
0  foo    7
1  foo    8
2  bar    7
3  bar    8
```

3.2. General functions
pandas.merge_ordered

 pandas.merge_ordered(left, right=None, on=None, left_on=None, right_on=None, left_by=None, right_by=None, fill_method=None, suffixes=None, how='outer')  

Perform a merge for ordered data with optional filling/interpolation.

Designed for ordered data like time series data. Optionally perform group-wise merge (see examples).

Parameters

left  [DataFrame]  
right [DataFrame]  
on [label or list] Field names to join on. Must be found in both DataFrames.
left_on [label or list, or array-like] Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns.
right_on [label or list, or array-like] Field names to join on in right DataFrame or vector/list of vectors per left_on docs.
left_by [column name or list of column names] Group left DataFrame by group columns and merge piece by piece with right DataFrame.
right_by [column name or list of column names] Group right DataFrame by group columns and merge piece by piece with left DataFrame.
fill_method [{‘ffill’, None}, default None] Interpolation method for data.
suffixes [list-like, default is ("_x", "_y") ] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in left and right respectively. Pass a value of None instead of a string to indicate that the column name from left or right should be left as-is, with no suffix. At least one of the values must not be None.
how [{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘outer’]  

Returns

DataFrame The merged DataFrame output type will the be same as ‘left’, if it is a subclass of DataFrame.

See also:

merge Merge with a database-style join.
merge_asof Merge on nearest keys.
Examples

```python
>>> df1 = pd.DataFrame(
...     {
...         "key": ["a", "c", "e", "a", "c", "e"],
...         "lvalue": [1, 2, 3, 1, 2, 3],
...         "group": ["a", "a", "a", "b", "b", "b"]
...     }
... )
>>> df1
key  lvalue group
0  a     1   a
1  c     2   a
2  e     3   a
3  a     1   b
4  c     2   b
5  e     3   b

>>> df2 = pd.DataFrame({"key": ["b", "c", "d"], "rvalue": [1, 2, 3]})
>>> df2
key  rvalue
0  b     1
1  c     2
2  d     3

>>> merge_ordered(df1, df2, fill_method="ffill", left_by="group")
key  lvalue group  rvalue
0  a     1   a     NaN
1  b     1   a     1.0
2  c     2   a     2.0
3  d     2   a     3.0
4  e     3   a     3.0
5  a     1   b     NaN
6  b     1   b     1.0
7  c     2   b     2.0
8  d     2   b     3.0
9  e     3   b     3.0
```

### pandas.merge_asof

**pandas.merge_asof**(left, right, on=None, left_on=None, right_on=None, left_index=False, right_index=False, by=None, left_by=None, right_by=None, suffixes=('_x', '_y'), tolerance=None, allow_exact_matches=True, direction='backward')

Perform a merge by key distance.

This is similar to a left-join except that we match on nearest key rather than equal keys. Both DataFrames must be sorted by the key.

For each row in the left DataFrame:

- A “backward” search selects the last row in the right DataFrame whose ‘on’ key is less than or equal to the left’s key.
• A “forward” search selects the first row in the right DataFrame whose ‘on’ key is greater than or equal to the left’s key.
• A “nearest” search selects the row in the right DataFrame whose ‘on’ key is closest in absolute distance to the left’s key.

The default is “backward” and is compatible in versions below 0.20.0. The direction parameter was added in version 0.20.0 and introduces “forward” and “nearest”.

Optionally match on equivalent keys with ‘by’ before searching with ‘on’.

Parameters

left  [DataFrame or named Series]
right [DataFrame or named Series]
on  [label] Field name to join on. Must be found in both DataFrames. The data MUST be ordered. Furthermore this must be a numeric column, such as datetimelike, integer, or float. On or left_on/right_on must be given.
left_on  [label] Field name to join on in left DataFrame.
right_on [label] Field name to join on in right DataFrame.
left_index  [bool] Use the index of the left DataFrame as the join key.
right_index [bool] Use the index of the right DataFrame as the join key.
by  [column name or list of column names] Match on these columns before performing merge operation.
left_by  [column name] Field names to match on in the left DataFrame.
right_by [column name] Field names to match on in the right DataFrame.
suffixes  [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively.
tolerance  [int or Timedelta, optional, default None] Select asof tolerance within this range; must be compatible with the merge index.
allow_exact_matches  [bool, default True]
  • If True, allow matching with the same ‘on’ value (i.e. less-than-or-equal-to / greater-than-or-equal-to)
  • If False, don’t match the same ‘on’ value (i.e., strictly less-than / strictly greater-than).
direction  ['backward' (default), ‘forward’, or ‘nearest’] Whether to search for prior, subsequent, or closest matches.

Returns

merged  [DataFrame]

See also:
merge  Merge with a database-style join.
merge_ordered  Merge with optional filling/interpolation.
Examples

```python
>>> left = pd.DataFrame({"a": [1, 5, 10], "left_val": ["a", "b", "c"]})
>>> left
   a         left_val
0  1         a
1  5         b
2 10         c

>>> right = pd.DataFrame({"a": [1, 2, 3, 6, 7], "right_val": [1, 2, 3, 6, 7]})
>>> right
   a    right_val
0  1      1
1  2      2
2  3      3
3  6      6
4  7      7

>>> pd.merge_asof(left, right, on="a")
   a left_val right_val
0  1      a       1
1  5      b       3
2 10     c       NaN

>>> pd.merge_asof(left, right, on="a", allow_exact_matches=False)
   a left_val right_val
0  1      a       NaN
1  5      b       3.0
2 10     c       7.0

>>> pd.merge_asof(left, right, on="a", direction="forward")
   a left_val right_val
0  1      a       1.0
1  5      b       6.0
2 10     c       NaN

>>> pd.merge_asof(left, right, on="a", direction="nearest")
   a left_val right_val
0  1      a       1
1  5      b       6
2 10     c       7

We can use indexed DataFrames as well.

```python
>>> left = pd.DataFrame({"left_val": ["a", "b", "c"], index=[1, 5, 10]})
>>> left
   left_val
1     a
5     b
10    c
```
>>> right = pd.DataFrame(        "right_val": [1, 2, 3, 6, 7],        index=[1, 2, 3, 6, 7])
>>> right
   right_val
1      1
2      2
3      3
6      6
7      7

>>> pd.merge_asof(left, right, left_index=True, right_index=True)
   left_val  right_val
0        a       1
1        b       3
2       c       7

Here is a real-world times-series example

>>> quotes = pd.DataFrame(
...     {        "time": [pd.Timestamp("2016-05-25 13:30:00.023"), pd.Timestamp("2016-05-25 13:30:00.023"),
...         pd.Timestamp("2016-05-25 13:30:00.030"), pd.Timestamp("2016-05-25 13:30:00.041"),
...         pd.Timestamp("2016-05-25 13:30:00.048"), pd.Timestamp("2016-05-25 13:30:00.049"),
...         pd.Timestamp("2016-05-25 13:30:00.072"), pd.Timestamp("2016-05-25 13:30:00.075")],
...     "ticker": ["GOOG", "MSFT", "MSFT", "MSFT", "GOOG", "AAPL", "GOOG", "MSFT"],
...     "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.01],
...     "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.03]
... }
... )
>>> quotes
       time  ticker  bid  ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT  51.95   51.96
2 2016-05-25 13:30:00.030   MSFT  51.97   51.98
3 2016-05-25 13:30:00.041   MSFT  51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL  97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT  52.01   52.03
By default we are taking the asof of the quotes:

```python
>>> pd.merge_asof(trades, quotes, on="time", by="ticker")
```

<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2016-05-25 13:30:00.038</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 2016-05-25 13:30:00.048</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We only asof within 2ms between the quote time and the trade time:

```python
>>> pd.merge_asof(trades, quotes, on="time", by="ticker", tolerance=pd.Timedelta("2ms"))
```

<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2016-05-25 13:30:00.038</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 2016-05-25 13:30:00.048</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. However prior data will propagate forward:

```python
>>> pd.merge_asof(trades, quotes, on="time", by="ticker", tolerance=pd.Timedelta("10ms"))
```

<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2016-05-25 13:30:00.038</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 2016-05-25 13:30:00.048</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
allow_exact_matches=False

<table>
<thead>
<tr>
<th>time</th>
<th>ticker</th>
<th>price</th>
<th>quantity</th>
<th>bid</th>
<th>ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2016-05-25 13:30:00.023</td>
<td>MSFT</td>
<td>51.95</td>
<td>75</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>2016-05-25 13:30:00.038</td>
<td>MSFT</td>
<td>51.95</td>
<td>155</td>
<td>51.97</td>
</tr>
<tr>
<td>2</td>
<td>2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.77</td>
<td>100</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>2016-05-25 13:30:00.048</td>
<td>GOOG</td>
<td>720.92</td>
<td>100</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>2016-05-25 13:30:00.048</td>
<td>AAPL</td>
<td>98.00</td>
<td>100</td>
<td>NaN</td>
</tr>
</tbody>
</table>

pandas.concat

pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)

Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

Parameters

- **objs** [a sequence or mapping of Series or DataFrame objects] If a mapping is passed, the sorted keys will be used as the keys argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.
- **axis** [{0/'index', 1/'columns'}, default 0] The axis to concatenate along.
- **join** [{‘inner’, ‘outer’}, default ‘outer’] How to handle indexes on other axis (or axes).
- **ignore_index** [bool, default False] If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.
- **keys** [sequence, default None] If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level.
- **levels** [list of sequences, default None] Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.
- **names** [list, default None] Names for the levels in the resulting hierarchical index.
- **verify_integrity** [bool, default False] Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.
- **sort** [bool, default False] Sort non-concatenation axis if it is not already aligned when join is ‘outer’. This has no effect when join=’inner’, which already preserves the order of the non-concatenation axis.

Changed in version 1.0.0: Changed to not sort by default.

**copy** [bool, default True] If False, do not copy data unnecessarily.

Returns

**object, type of objs** When concatenating all Series along the index (axis=0), a Series is returned. When objs contains at least one DataFrame, a DataFrame is returned. When concatenating along the columns (axis=1), a DataFrame is returned.
See also:

*Series.append*  Concatenate Series.

*DataFrame.append*  Concatenate DataFrames.

*DataFrame.join*  Join DataFrames using indexes.

*DataFrame.merge*  Merge DataFrames by indexes or columns.

Notes

The keys, levels, and names arguments are all optional.

A walkthrough of how this method fits in with other tools for combining pandas objects can be found [here](#).

Examples

Combine two Series.

```python
>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```

Clear the existing index and reset it in the result by setting the `ignore_index` option to True.

```python
>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object
```

Add a hierarchical index at the outermost level of the data with the `keys` option.

```python
>>> pd.concat([s1, s2], keys=['s1', 's2'])
s1  0    a
    1    b
s2  0    c
    1    d
dtype: object
```

Label the index keys you create with the `names` option.

```python
>>> pd.concat([s1, s2], keys=['s1', 's2'], names=['Series name', 'Row ID'])
Series name    Row ID
s1  0    a
    1    b
s2  0    c
```

(continues on next page)
1 d
dtype: object

Combine two DataFrame objects with identical columns.

```python
>>> df1 = pd.DataFrame([
    ['a', 1], ['b', 2],
    ...
    columns=['letter', 'number'])

>>> df1
letter   number
0   a       1
1   b       2

>>> df2 = pd.DataFrame([
    ['c', 3], ['d', 4],
    ...
    columns=['letter', 'number'])

>>> df2
letter   number
0   c       3
1   d       4
```

```python
>>> pd.concat([df1, df2])
letter   number
0   a       1
1   b       2
0   c       3
1   d       4
```

Combine DataFrame objects with overlapping columns and return everything. Columns outside the intersection will be filled with NaN values.

```python
>>> df3 = pd.DataFrame([
    ['c', 3, 'cat'], ['d', 4, 'dog'],
    ...
    columns=['letter', 'number', 'animal'])

>>> df3
letter   number   animal
0   c       3       cat
1   d       4       dog
```

```python
>>> pd.concat([df1, df3], sort=False)
letter   number   animal
0   a       1       NaN
1   b       2       NaN
0   c       3       cat
1   d       4       dog
```

Combine DataFrame objects with overlapping columns and return only those that are shared by passing inner to the join keyword argument.

```python
>>> pd.concat([df1, df3], join="inner")
letter   number
0   a       1
1   b       2
0   c       3
1   d       4
```

Combine DataFrame objects horizontally along the x axis by passing in axis=1.
>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                     columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
letter  number  animal  name
0    a      1   bird   polly
1    b      2  monkey  george

Prevent the result from including duplicate index values with the verify_integrity option.

>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
0
a 1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
0
a 2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: ['a']

pandas.get_dummies

pandas.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)

Convert categorical variable into dummy/indicator variables.

Parameters

data  [array-like, Series, or DataFrame] Data of which to get dummy indicators.

prefix  [str, list of str, or dict of str, default None] String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, prefix can be a dictionary mapping column names to prefixes.

prefix_sep  [str, default '_'] If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with prefix.

dummy_na  [bool, default False] Add a column to indicate NaNs, if False NaNs are ignored.

columns  [list-like, default None] Column names in the DataFrame to be encoded. If columns is None then all the columns with object or category dtype will be converted.

sparse  [bool, default False] Whether the dummy-encoded columns should be backed by a SparseArray (True) or a regular NumPy array (False).

drop_first  [bool, default False] Whether to get k-1 dummies out of k categorical levels by removing the first level.

dtype  [dtype, default np.uint8] Data type for new columns. Only a single dtype is allowed.

Returns

DataFrame  Dummy-coded data.

See also:
Series.str.get_dummies  Convert Series to dummy codes.

Notes

Reference the user guide for more examples.

Examples

```python
>>> s = pd.Series(list('abca'))

>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0

>>> s1 = ['a', 'b', np.nan]

>>> pd.get_dummies(s1)
   a  b
0  1  0
1  0  1
2  0  0

>>> pd.get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0  0
1  0  1  0
2  0  0  1

>>> df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
                    'C': [1, 2, 3]})

>>> pd.get_dummies(df, prefix=['col1', 'col2'])
    C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1  1  0  0  0  1  0
1  2  0  1  1  0  0
2  3  1  0  0  0  1

>>> pd.get_dummies(pd.Series(list('abca')))   
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0
```
```python
>>> pd.get_dummies(pd.Series(list('abca')), drop_first=True)
b c
0 0 0
1 1 0
2 0 1
3 0 0
4 0 0
```

```python
>>> pd.get_dummies(pd.Series(list('abc')), dtype=float)
a b c
0 1.0 0.0 0.0
1 0.0 1.0 0.0
2 0.0 0.0 1.0
```

**pandas.factorize**

`pandas.factorize(values, sort=False, na_sentinel=-1, size_hint=None)`

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. `factorize` is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

**Parameters**

- `values` [sequence] A 1-D sequence. Sequences that aren’t pandas objects are coerced to ndarrays before factorization.
- `sort` [bool, default False] Sort `uniques` and shuffle `codes` to maintain the relationship.
- `na_sentinel` [int or None, default -1] Value to mark “not found”. If None, will not drop the NaN from the uniques of the values.

  Changed in version 1.1.2.
- `size_hint` [int, optional] Hint to the hashtable sizer.

**Returns**

- `codes` [ndarray] An integer ndarray that’s an indexer into `uniques`. `uniques.take(codes)` will have the same values as `values`.
- `uniques` [ndarray, Index, or Categorical] The unique valid values. When `values` is Categorical, `uniques` is a Categorical. When `values` is some other pandas object, an `Index` is returned. Otherwise, a 1-D ndarray is returned.

  **Note:** Even if there’s a missing value in `values`, `uniques` will not contain an entry for it.

**See also:**

- `cut` Discretize continuous-valued array.
- `unique` Find the unique value in an array.
Notes

Reference the user guide for more examples.

Examples

These examples all show factorize as a top-level method like pd.factorize(values). The results are identical for methods like Series.factorize().

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> codes
array([0, 0, 1, 2, 0]...)
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With sort=True, the uniques will be sorted, and codes will be shuffled so that the relationship is the maintained.

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> codes
array([1, 1, 0, 2, 1]...)
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in codes with na_sentinel (-1 by default). Note that missing values are never included in uniques.

```python
>>> codes, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> codes
array([ 0, -1, 1, 2, 0]...)
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we’ve only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of uniques will differ. For Categoricals, a Categorical is returned.

```python
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1]...)
>>> uniques
['a', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Notice that 'b' is in uniques.categories, despite not being present in cat.values.

For all other pandas objects, an Index of the appropriate type is returned.

```python
>>> cat = pd.Series(['a', 'a', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1]...)
>>> uniques
Index(['a', 'c'], dtype='object')
```
If NaN is in the values, and we want to include NaN in the uniques of the values, it can be achieved by setting `na_sentinel=None`.

```python
>>> values = np.array([1, 2, 1, np.nan])
>>> codes, uniques = pd.factorize(values)  # default: na_sentinel=-1
>>> codes
array([ 0, 1, 0, -1])
>>> uniques
array([1., 2.])
>>> codes, uniques = pd.factorize(values, na_sentinel=None)
>>> codes
array([0, 1, 0, 2])
>>> uniques
array([ 1., 2., nan])
```

### pandas.unique

**pandas.unique(values)**

Return unique values based on a hash table.

Uniques are returned in order of appearance. This does NOT sort.

Significantly faster than numpy.unique for long enough sequences. Includes NA values.

**Parameters**

- **values** [1d array-like]

**Returns**

**numpy.ndarray or ExtensionArray** The return can be:

- **Index** : when the input is an Index
- **Categorical** : when the input is a Categorical dtype
- **ndarray** : when the input is a Series/ndarray

Return numpy.ndarray or ExtensionArray.

**See also:**

- **Index.unique** Return unique values from an Index.
- **Series.unique** Return unique values of Series object.

**Examples**

```python
>>> pd.unique(pd.Series([2, 1, 3, 3]))
array([2, 1, 3])
>>> pd.unique(pd.Series([2] + [1] * 5))
array([2, 1])
>>> pd.unique(pd.Series([pd.Timestamp("20160101"), pd.Timestamp("20160101")]))
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```
```python
>>> pd.unique(
...     pd.Series(
...         [pd.Timestamp("20160101", tz="US/Eastern"),
...          pd.Timestamp("20160101", tz="US/Eastern"),
...          ]
...     )
... )
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

>>> pd.unique(
...     pd.Index(
...         [pd.Timestamp("20160101", tz="US/Eastern"),
...          pd.Timestamp("20160101", tz="US/Eastern"),
...          ]
...     )
... )
DatetimeIndex(['2016-01-01 00:00:00-05:00'],
               dtype='datetime64[ns, US/Eastern]',
               freq=None)

>>> pd.unique(list("baabc"))
anarray(['b', 'a', 'c'], dtype=object)

An unordered Categorical will return categories in the order of appearance.

>>> pd.unique(pd.Series(pd.Categorical(list("baabc"))))
['b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> pd.unique(pd.Series(pd.Categorical(list("baabc"), categories=list("abc"))))
['b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']

An ordered Categorical preserves the category ordering.

>>> pd.unique(
...     pd.Series(
...         pd.Categorical(list("baabc"), categories=list("abc"), ordered=True)
...     )
... )
['b', 'a', 'c']
Categories (3, object): ['a' < 'b' < 'c']

An array of tuples

>>> pd.unique([("a", "b"), ("b", "a"), ("a", "c"), ("b", "a"),])
anarray([('a', 'b'), ('b', 'a'), ('a', 'c')], dtype=object)
```
pandas.wide_to_long

pandas.wide_to_long(df, stubnames, i, j, sep='', suffix='\d+')

Unpivot a DataFrame from wide to long format.

Less flexible but more user-friendly than melt.

With stubnames ['A', 'B'], this function expects to find one or more group of columns with format A-suffix1, A-suffix2,..., B-suffix1, B-suffix2,... You specify what you want to call this suffix in the resulting long format with j (for example j='year')

Each row of these wide variables are assumed to be uniquely identified by i (can be a single column name or a list of column names)

All remaining variables in the data frame are left intact.

Parameters

- df [DataFrame] The wide-format DataFrame.
- stubnames [str or list-like] The stub name(s). The wide format variables are assumed to start with the stub names.
- i [str or list-like] Column(s) to use as id variable(s).
- j [str] The name of the sub-observation variable. What you wish to name your suffix in the long format.
- sep [str, default ''] A character indicating the separation of the variable names in the wide format, to be stripped from the names in the long format. For example, if your column names are A-suffix1, A-suffix2, you can strip the hyphen by specifying sep='\-'.
- suffix [str, default '\d+'] A regular expression capturing the wanted suffixes. '\d+' captures numeric suffixes. Suffixes with no numbers could be specified with the negated character class '\D+'. You can also further disambiguate suffixes, for example, if your wide variables are of the form A-one, B-two,... and you have an unrelated column A-rating, you can ignore the last one by specifying suffix='(!?one|two)'. When all suffixes are numeric, they are cast to int64/float64.

Returns

- DataFrame A DataFrame that contains each stub name as a variable, with new index (i, j).

See also:

- melt Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
- pivot Create a spreadsheet-style pivot table as a DataFrame.
- DataFrame.pivot Pivot without aggregation that can handle non-numeric data.
- DataFrame.pivot_table Generalization of pivot that can handle duplicate values for one index/column pair.
- DataFrame.unstack Pivot based on the index values instead of a column.
Notes

All extra variables are left untouched. This simply uses pandas.melt under the hood, but is hard-coded to “do the right thing” in a typical case.

Examples

```python
>>> np.random.seed(123)
>>> df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
                      "A1980" : {0 : "d", 1 : "e", 2 : "f"},
                      "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
                      "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
                      "X" : dict(zip(range(3), np.random.randn(3)))
                      })
>>> df["id"] = df.index
>>> df
0    a     d    2.5    3.2  -1.085631  0
1    b     e    1.2    1.3   0.997345  1
2    c     f    0.7    0.1   0.282978  2
>>> pd.wide_to_long(df, ["A", "B"], i="id", j="year")
...   X  A  B
id year
0  1970  -1.085631  a  2.5
1  1970   0.997345  b  1.2
2  1970   0.282978  c  0.7
0  1980  -1.085631  d  3.2
1  1980   0.997345  e  1.3
2  1980   0.282978  f  0.1

With multiple id columns

```python
>>> df = pd.DataFrame({
                     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
                     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
                     'ht1': [2.8, 2.9, 2.2, 1.8, 1.9, 2.2, 2.3, 2.1],
                     'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
                     })
>>> df
  famid  birth  ht1  ht2
0   1     1    2.8  3.4
1   1     2    2.9  3.8
2   1     3    2.2  2.9
3   2     1    2.0  3.2
4   2     2    1.8  2.8
5   2     3    1.9  2.4
6   3     1    2.2  3.3
7   3     2    2.3  3.4
8   3     3    2.1  2.9
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age')
>>> l
```
...     h1   h2
famid birth age   
1  1  1  2.8  3.4
  2  1  2.9  3.8
  3  1  2.2  2.9
2  1  2.0  3.2
  2  1  1.8  2.8
  3  1  1.9  2.4
3  1  1  2.2  3.3
  2  1  2.3  3.4
  3  1  2.1  2.4
  2  1  2.0  3.2
  2  2  3.2
  2  1  2.9

Going from long back to wide just takes some creative use of `unstack`

```python
>>> w = l.unstack()
>>> w.columns = w.columns.map('{{0[0]}}{{0[1]}}'.format)
>>> w.reset_index()
   famid  birth  h1  h2
0   0     1     2.8  3.4
1   1     2     2.9  3.8
2   2     3     2.2  2.9
3   3     1     2.0  3.2
4   4     2     1.8  2.8
5   5     3     1.9  2.4
6   6     2     2.3  3.3
7   7     3     2.3  3.4
8   8     3     2.1  2.9
```

Less wieldy column names are also handled

```python
>>> np.random.seed(0)
>>> df = pd.DataFrame({'A(weekly)-2010': np.random.rand(3),
                     ... 'A(weekly)-2011': np.random.rand(3),
                     ... 'B(weekly)-2010': np.random.rand(3),
                     ... 'B(weekly)-2011': np.random.rand(3),
                     ... 'X': np.random.randint(3, size=3)})
>>> df['id'] = df.index
>>> df
   A(weekly)-2010  A(weekly)-2011  B(weekly)-2010  B(weekly)-2011    X  id
0  0.548814     0.544883     0.437587     0.383442  0  0
1  0.715189     0.423655     0.891773     0.791725  1  1
2  0.602763     0.645894     0.963663     0.528895  1  2
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> pd.wide_to_long(df, ['A(weekly)', 'B(weekly)'], i='id',
...     j='year', sep='-')
...     X A(weekly) B(weekly)
id year
0 2010 0 0.548814 0.437587
1 2010 1 0.715189 0.891773
2 2010 1 0.602763 0.963663
0 2011 0 0.544883 0.383442
1 2011 1 0.423655 0.791725
2 2011 1 0.645894 0.528895
```

If we have many columns, we could also use a regex to find our stubnames and pass that list on to wide_to_long

```python
>>> stubnames = sorted(
...     set([match[0] for match in df.columns.str.findall(  
...         r'\[A-B]\(.*\)', values if match != []]))
... )
>>> list(stubnames)
['A(weekly)', 'B(weekly)']
```

All of the above examples have integers as suffixes. It is possible to have non-integers as suffixes.

```python
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht_one': [2.8, 2.9, 2.2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht_two': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
      famid  birth  ht_one  ht_two
0       1       1     2.8     3.4
1       1       2     2.9     3.8
2       1       3     2.2     2.9
3       2       1     2.0     3.2
4       2       2     1.8     2.8
5       3       1     2.2     3.3
6       3       2     2.3     3.4
7       3       3     2.1     2.9
8       3       3     2.1     2.9

>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age',
...     sep='_', suffix=r'\w+', )
>>> l

```

(continues on next page)
2 1 one 2.0
two 3.2
2 one 1.8
two 2.8
3 one 1.9
two 2.4
3 1 one 2.2
two 3.3
2 one 2.3
two 3.4
3 one 2.1
two 2.9

### 3.2.2 Top-level missing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isna(obj)</code></td>
<td>Detect missing values for an array-like object.</td>
</tr>
<tr>
<td><code>isnull(obj)</code></td>
<td>Detect missing values for an array-like object.</td>
</tr>
<tr>
<td><code>notna(obj)</code></td>
<td>Detect non-missing values for an array-like object.</td>
</tr>
<tr>
<td><code>notnull(obj)</code></td>
<td>Detect non-missing values for an array-like object.</td>
</tr>
</tbody>
</table>

**pandas.isna**

pandas.

`.isna(obj)`

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (\(NaN\) in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

**Parameters**

- `obj` [scalar or array-like] Object to check for null or missing values.

**Returns**

- `bool` or `array-like of bool` For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

**See also:**

- `notna` Boolean inverse of pandas.isna.
- `Series.isna` Detect missing values in a Series.
- `DataFrame.isna` Detect missing values in a DataFrame.
- `Index.isna` Detect missing values in an Index.
Examples

Scalar arguments (including strings) result in a scalar boolean.

```python
>>> pd.isna('dog')
False
```

```python
>>> pd.isna(pd.NA)
True
```

```python
>>> pd.isna(np.nan)
True
```

Ndarrays result in an ndarray of booleans.

```python
array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
array
```

```python
array([[ 1., nan, 3.],
       [ 4., 5., nan]])
```

```python
pd.isna(array)
```

```python
array([[False, True, False],
       [False, False, True]])
```

For indexes, an ndarray of booleans is returned.

```python
index = pd.DatetimeIndex(['2017-07-05', '2017-07-06', None,...
                           '2017-07-08'])
index
```

```python
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
               dtype='datetime64[ns]', freq=None)
```

```python
pd.isna(index)
```

```python
array([False, False, True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```python
df = pd.DataFrame([[ant, bee, cat], [dog, None, fly]])
```

```python
df
```

```python
0     1     2
0    ant   bee   cat
1     dog  None    fly
```

```python
pd.isna(df)
```

```python
0     1     2
0 False False False
1 False  True False
```

```python
pd.isna(df[1])
```

```python
0 False
1 True
Name: 1, dtype: bool
```
pandas.isnull

pandas.isnull(obj)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

obj [scalar or array-like] Object to check for null or missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

notna Boolean inverse of pandas.isna.
Series.isna Detect missing values in a Series.
DataFrame.isna Detect missing values in a DataFrame.
Index.isna Detect missing values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```python
>>> pd.isna('dog')
False
```

```python
>>> pd.isna(pd.NA)
True
```

```python
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```python
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan, 3.],
       [ 4., 5., nan]])
>>> pd.isna(array)
array([[False, True, False],
       [False, False, True]])
```

For indexes, an ndarray of booleans is returned.

```python
>>> index = pd.DatetimeIndex(['2017-07-05', '2017-07-06', None, ...
                          '2017-07-08'])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
```
```python
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```python
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
  0  1  2
0 ant bee cat
1 dog None fly
```

```python
>>> pd.isna(df)
   0   1   2
0  False  False  False
1  False   True  False
```

```python
>>> pd.isna(df[1])
0   False
1    True
Name: 1, dtype: bool
```

### `pandas.notna`

`pandas.notna(obj)`

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is `NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

**Parameters**

- **obj** [array-like or object value] Object to check for not null or non-missing values.

**Returns**

- **bool or array-like of bool** For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

**See also:**

- `isna` Boolean inverse of `pandas.notna`.
- `Series.notna` Detect valid values in a Series.
- `DataFrame.notna` Detect valid values in a DataFrame.
- `Index.notna` Detect valid values in an Index.
**Examples**

Scalar arguments (including strings) result in a scalar boolean.

```python
>>> pd.notna('dog')
True
```

```python
>>> pd.notna(pd.NA)
False
```

```python
>>> pd.notna(np.nan)
False
```

Ndarrays result in an ndarray of booleans.

```python
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan, 3.],
       [ 4., 5., nan]])
>>> pd.notna(array)
array([[ True, False, True],
       [ True, True, False]])
```

For indexes, an ndarray of booleans is returned.

```python
>>> index = pd.DatetimeIndex(['2017-07-05', '2017-07-06', None, ...
                              '2017-07-08'])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True, True, False, True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```python
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
   0  1  2
0  ant bee cat
1  dog None fly
>>> pd.notna(df)
   0  1  2
0  True True True
1  True False True
>>> pd.notna(df[1])
   0   1
0  True
1  False
Name: 1, dtype: bool
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.notnull

pandas.notnull(obj)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

- obj [array-like or object value] Object to check for not null or non-missing values.

Returns

- bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

See also:

- isna Boolean inverse of pandas.notna.
- Series.notna Detect valid values in a Series.
- DataFrame.notna Detect valid values in a DataFrame.
- Index.notna Detect valid values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```python
>>> pd.notna('dog')
True
```

```python
>>> pd.notna(pd.NA)
False
```

```python
>>> pd.notna(np.nan)
False
```

Ndarrays result in an ndarray of booleans.

```python
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
```

```python
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
```

```python
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```python
>>> index = pd.DatetimeIndex(['2017-07-05', '2017-07-06', None,
                               ..., '2017-07-08'])
```

```python
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
               dtype='datetime64[ns]', freq=None)
```
```python
>>> pd.notna(index)
array([ True, True, False, True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```python
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
      0  1  2
0  ant  bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0  1  2
0  True True True
1  True False True
```

```python
>>> pd.notna(df[1])
0  True
1  False
Name: 1, dtype: bool
```

### 3.2.3 Top-level dealing with numeric data

**to_numeric**(arg[, errors, downcast])

Convert argument to a numeric type.

**pandas.to_numeric**

```python
pandas.to_numeric(arg, errors='raise', downcast=None)
```

Convert argument to a numeric type.

The default return dtype is float64 or int64 depending on the data supplied. Use the `downcast` parameter to obtain other dtypes.

Please note that precision loss may occur if really large numbers are passed in. Due to the internal limitations of ndarray, if numbers smaller than -9223372036854775808 (np.iinfo(np.int64).min) or larger than 18446744073709551615 (np.iinfo(np.uint64).max) are passed in, it is very likely they will be converted to float so that they can stored in an ndarray. These warnings apply similarly to Series since it internally leverages ndarray.

**Parameters**

- `arg` [scalar, list, tuple, 1-d array, or Series] Argument to be converted.
- `errors` ['ignore', 'raise', 'coerce'], default ‘raise’
  - If ‘raise’, then invalid parsing will raise an exception.
  - If ‘coerce’, then invalid parsing will be set as NaN.
  - If ‘ignore’, then invalid parsing will return the input.
- `downcast` [str, default None] Can be ‘integer’, ‘signed’, ‘unsigned’, or ‘float’. If not None, and if the data has been successfully cast to a numerical dtype (or if the data was numeric to begin with), downcast that resulting data to the smallest numerical dtype possible according to the following rules:
pandas: powerful Python data analysis toolkit, Release 1.4.2

- ‘integer’ or ‘signed’: smallest signed int dtype (min.: np.int8)
- ‘unsigned’: smallest unsigned int dtype (min.: np.uint8)
- ‘float’: smallest float dtype (min.: np.float32)

As this behaviour is separate from the core conversion to numeric values, any errors raised during the downcasting will be surfaced regardless of the value of the ‘errors’ input.

In addition, downcasting will only occur if the size of the resulting data’s dtype is strictly larger than the dtype it is to be cast to, so if none of the dtypes checked satisfy that specification, no downcasting will be performed on the data.

Returns

- **ret** Numeric if parsing succeeded. Return type depends on input. Series if Series, otherwise ndarray.

See also:

- **DataFrame.astype** Cast argument to a specified dtype.
- **to_datetime** Convert argument to datetime.
- **to_timedelta** Convert argument to timedelta.
- **numpy.ndarray.astype** Cast a numpy array to a specified type.
- **DataFrame.convert_dtypes** Convert dtypes.

Examples

Take separate series and convert to numeric, coercing when told to

```python
>>> s = pd.Series(['1.0', '2', -3])
>>> pd.to_numeric(s)
0    1.0
1    2.0
2   -3.0
dtype: float64

>>> pd.to_numeric(s, downcast='float')
0    1.0
1    2.0
2   -3.0
dtype: float32

>>> pd.to_numeric(s, downcast='signed')
0    1
1    2
2    -3
dtype: int8

>>> s = pd.Series(['apple', '1.0', '2', -3])
>>> pd.to_numeric(s, errors='ignore')
apple
0    1.0
1    2
2    -3
dtype: object

>>> pd.to_numeric(s, errors='coerce')
```

(continues on next page)
Downcasting of nullable integer and floating dtypes is supported:

```
>>> s = pd.Series([1, 2, 3], dtype="Int64")
>>> pd.to_numeric(s, downcast="integer")
0 1
1 2
2 3
dtype: Int8
```

```
>>> s = pd.Series([1.0, 2.1, 3.0], dtype="Float64")
>>> pd.to_numeric(s, downcast="float")
0 1.0
1 2.1
2 3.0
dtype: Float32
```

### 3.2.4 Top-level dealing with datetimelike data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_datetime</code></td>
<td>Convert argument to datetime.</td>
</tr>
<tr>
<td><code>to_timedelta</code></td>
<td>Convert argument to timedelta.</td>
</tr>
<tr>
<td><code>date_range</code></td>
<td>Return a fixed frequency DatetimeIndex.</td>
</tr>
<tr>
<td><code>bdate_range</code></td>
<td>Return a fixed frequency DatetimeIndex, with business day as the default frequency.</td>
</tr>
<tr>
<td><code>period_range</code></td>
<td>Return a fixed frequency PeriodIndex.</td>
</tr>
<tr>
<td><code>timedelta_range</code></td>
<td>Return a fixed frequency TimedeltaIndex, with day as the default frequency.</td>
</tr>
<tr>
<td><code>infer_freq</code></td>
<td>Infer the most likely frequency given the input index.</td>
</tr>
</tbody>
</table>

**pandas.to_datetime**

`pandas.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, format=None, exact=True, unit=None, infer_datetime_format=False, origin='unix', cache=True)`

Convert argument to datetime.

This function converts a scalar, array-like, `Series` or `DataFrame/dict-like` to a pandas datetime object.

**Parameters**

- **arg** [int, float, str, datetime, list, tuple, 1-d array, Series, DataFrame/dict-like] The object to convert to a datetime. If a `DataFrame` is provided, the method expects minimally the following columns: "year", "month", "day".
- **errors** [{‘ignore’, ‘raise’, ‘coerce’}, default ‘raise’]
  - If 'raise', then invalid parsing will raise an exception.
  - If 'coerce', then invalid parsing will be set as NaT.
• If 'ignore', then invalid parsing will return the input.

dayfirst [bool, default False] Specify a date parse order if arg is str or is list-like. If True, parses dates with the day first, e.g. "10/11/12" is parsed as 2012-11-10.

**Warning:** dayfirst=True is not strict, but will prefer to parse with day first. If a delimited date string cannot be parsed in accordance with the given dayfirst option, e.g. to_datetime(['31-12-2021']), then a warning will be shown.

yearfirst [bool, default False] Specify a date parse order if arg is str or is list-like.

• If True parses dates with the year first, e.g. "10/11/12" is parsed as 2010-11-12.
• If both dayfirst and yearfirst are True, yearfirst is preceded (same as dateutil).

**Warning:** yearfirst=True is not strict, but will prefer to parse with year first.

utc [bool, default None] Control timezone-related parsing, localization and conversion.

• If True, the function always returns a timezone-aware UTC-localized Timestamp, Series or DatetimeIndex. To do this, timezone-naive inputs are localized as UTC, while timezone-aware inputs are converted to UTC.
• If False (default), inputs will not be coerced to UTC. Timezone-naive inputs will remain naive, while timezone-aware ones will keep their time offsets. Limitations exist for mixed offsets (typically, daylight savings), see Examples section for details.

See also: pandas general documentation about timezone conversion and localization.

format [str, default None] The strftime to parse time, e.g. "%d/%m/%Y". Note that "%f" will parse all the way up to nanoseconds. See strftime documentation for more information on choices.

exact [bool, default True] Control how format is used:

• If True, require an exact format match.
• If False, allow the format to match anywhere in the target string.

unit [str, default 'ns'] The unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer or float number. This will be based off the origin. Example, with unit='ms' and origin='unix' (the default), this would calculate the number of milliseconds to the unix epoch start.

infer_datetime_format [bool, default False] If True and no format is given, attempt to infer the format of the datetime strings based on the first non-NaN element, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

origin [scalar, default 'unix'] Define the reference date. The numeric values would be parsed as number of units (defined by unit) since this reference date.

• If 'unix' (or POSIX) time; origin is set to 1970-01-01.
• If 'julian', unit must be 'D', and origin is set to beginning of Julian Calendar. Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.
• If Timestamp convertible, origin is set to Timestamp identified by origin.
cache [bool, default True] If True, use a cache of unique, converted dates to apply the date-time conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets. The cache is only used when there are at least 50 values. The presence of out-of-bounds values will render the cache unusable and may slow down parsing.

Changed in version 0.25.0: changed default value from False to True.

Returns

datetime If parsing succeeded. Return type depends on input (types in parenthesis correspond to fallback in case of unsuccessful timezone or out-of-range timestamp parsing):

- scalar: Timestamp (or datetime.datetime)
- array-like: DatetimeIndex (or Series with object dtype containing datetime.datetime)
- Series: Series of datetime64 dtype (or Series of object dtype containing datetime.datetime)
- DataFrame: Series of datetime64 dtype (or Series of object dtype containing datetime.datetime)

Raises

ParserError When parsing a date from string fails.

ValueError When another datetime conversion error happens. For example when one of ‘year’, ‘month’, day’ columns is missing in a DataFrame, or when a Timezone-aware datetime.datetime is found in an array-like of mixed time offsets, and utc=False.

See also:

DataFrame.astype Cast argument to a specified dtype.
to_timedelta Convert argument to timedelta.
convert_dtypes Convert dtypes.

Notes

Many input types are supported, and lead to different output types:

- scalars can be int, float, str, datetime object (from stdlib datetime module or numpy). They are converted to Timestamp when possible, otherwise they are converted to datetime.datetime. None/NaN/null scalars are converted to NaT.

- array-like can contain int, float, str, datetime objects. They are converted to DatetimeIndex when possible, otherwise they are converted to Index with object dtype, containing datetime.datetime. None/NaN/null entries are converted to NaT in both cases.

- Series are converted to Series with datetime64 dtype when possible, otherwise they are converted to Series with object dtype, containing datetime.datetime. None/NaN/null entries are converted to NaT in both cases.

- DataFrame/dict-like are converted to Series with datetime64 dtype. For each row a datetime is created from assembling the various dataframe columns. Column keys can be common abbreviations like [‘year’, ‘month’, ‘day’, ‘minute’, ‘second’, ‘ms’, ‘us’, ‘ns’]) or plurals of the same.
The following causes are responsible for `datetime.datetime` objects being returned (possibly inside an `Index` or a `Series` with `object` dtype) instead of a proper pandas designated type (`Timestamp`, `DatetimeIndex` or `Series` with `datetime64` dtype):

- when any input element is before `Timestamp.min` or after `Timestamp.max`, see timestamp limitations.
- when `utc=False` (default) and the input is an array-like or `Series` containing mixed naive/aware datetime, or aware with mixed time offsets. Note that this happens in the (quite frequent) situation when the timezone has a daylight savings policy. In that case you may wish to use `utc=True`.

### Examples

#### Handling various input formats

Assembling a datetime from multiple columns of a `DataFrame`. The keys can be common abbreviations like (`'year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns'`) or plurals of the same

```python
>>> df = pd.DataFrame({'year': [2015, 2016],
...                    'month': [2, 3],
...                    'day': [4, 5]})
>>> pd.to_datetime(df)
0 2015-02-04
1 2016-03-05
dtype: datetime64[ns]
```

Passing `infer_datetime_format=True` can often-times speedup a parsing if its not an ISO8601 format exactly, but in a regular format.

```python
>>> s.head()
0 3/11/2000
1 3/12/2000
2 3/13/2000
3 3/11/2000
4 3/12/2000
dtype: object
```

```python
>>> %timeit pd.to_datetime(s, infer_datetime_format=True)
100 loops, best of 3: 10.4 ms per loop
```

```python
>>> %timeit pd.to_datetime(s, infer_datetime_format=False)
1 loop, best of 3: 471 ms per loop
```

Using a unix epoch time

```python
>>> pd.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
>>> pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

**Warning:** For float arg, precision rounding might happen. To prevent unexpected behavior use a fixed-width exact type.
Using a non-unix epoch origin

```python
>>> pd.to_datetime([1, 2, 3], unit='D',
...                  origin=pd.Timestamp('1960-01-01'))
DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'],
              dtype='datetime64[ns]', freq=None)
```

Non-convertible date/times

If a date does not meet the timestamp limitations, passing `errors='ignore'` will return the original input instead of raising any exception.

Passing `errors='coerce'` will force an out-of-bounds date to `NaT`, in addition to forcing non-dates (or non-parseable dates) to `NaT`.

```python
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='ignore')
datetime.datetime(1300, 1, 1, 0, 0)
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='coerce')
NaT
```

Timezones and time offsets

The default behaviour (`utc=False`) is as follows:

- Timezone-naive inputs are converted to timezone-naive `DatetimeIndex`:

```python
>>> pd.to_datetime(['2018-10-26 12:00', '2018-10-26 13:00:15'])
DatetimeIndex(['2018-10-26 12:00:00', '2018-10-26 13:00:15'],
              dtype='datetime64[ns]', freq=None)
```

- Timezone-aware inputs with constant time offset are converted to timezone-aware `DatetimeIndex`:

```
>>> pd.to_datetime(['2018-10-26 12:00 -0500', '2018-10-26 13:00 -0500'])
DatetimeIndex(['2018-10-26 12:00:00-05:00', '2018-10-26 13:00:00-05:00'],
              dtype='datetime64[ns, pytz.FixedOffset(-300)]', freq=None)
```

- However, timezone-aware inputs with mixed time offsets (for example issued from a timezone with daylight savings, such as Europe/Paris) are not successfully converted to a `DatetimeIndex`. Instead a simple `Index` containing `datetime.datetime` objects is returned:

```
>>> pd.to_datetime(['2020-10-25 02:00 +0200', '2020-10-25 04:00 +0100'])
Index([2020-10-25 02:00:00+02:00, 2020-10-25 04:00:00+01:00],
      dtype='object')
```

- A mix of timezone-aware and timezone-naive inputs is converted to a timezone-aware `DatetimeIndex` if the offsets of the timezone-aware are constant:

```python
>>> from datetime import datetime
>>> pd.to_datetime(['2020-01-01 01:00 -01:00', datetime(2020, 1, 1, 3, 0)])
DatetimeIndex(['2020-01-01 01:00:00-01:00', '2020-01-01 02:00:00-01:00'],
              dtype='datetime64[ns, pytz.FixedOffset(-60)]', freq=None)
```

- Finally, mixing timezone-aware strings and `datetime.datetime` always raises an error, even if the elements all have the same time offset.
```
>>> from datetime import datetime, timezone, timedelta

>>> d = datetime(2020, 1, 1, 18, tzinfo=timezone(-timedelta(hours=1)))

>>> pd.to_datetime(['2020-01-01 17:00 -0100', d])
Traceback (most recent call last):
...
ValueError: Tz-aware datetime.datetime cannot be converted to datetime64
unless utc=True
```

Setting `utc=True` solves most of the above issues:

- Timezone-naive inputs are localized as UTC

```
>>> pd.to_datetime(['2018-10-26 12:00', '2018-10-26 13:00'], utc=True)
DatetimeIndex(['2018-10-26 12:00:00+00:00', '2018-10-26 13:00:00+00:00'],
dtype='datetime64[ns, UTC]', freq=None)
```

- Timezone-aware inputs are converted to UTC (the output represents the exact same datetime, but viewed from the UTC time offset +00:00).

```
>>> pd.to_datetime(['2018-10-26 12:00 -0530', '2018-10-26 12:00 -0500'], utc=True)
DatetimeIndex(['2018-10-26 17:30:00+00:00', '2018-10-26 17:00:00+00:00'],
dtype='datetime64[ns, UTC]', freq=None)
```

- Inputs can contain both naive and aware, string or datetime, the above rules still apply

```
>>> pd.to_datetime(['2018-10-26 12:00', '2018-10-26 12:00 -0530',
...                 '2020-01-01 18:00:00+00:00', '2020-01-01 19:00:00+00:00'],
...                 utc=True)
DatetimeIndex(['2018-10-26 12:00:00+00:00', '2018-10-26 17:30:00+00:00',
...                 '2020-01-01 18:00:00+00:00', '2020-01-01 19:00:00+00:00'],
...                 dtype='datetime64[ns, UTC]', freq=None)
```

---

**pandas.to_timedelta**

`pandas.to_timedelta(arg, unit=None, errors='raise')`

Convert argument to timedelta.

Timedeltas are absolute differences in times, expressed in difference units (e.g. days, hours, minutes, seconds). This method converts an argument from a recognized timedelta format / value into a Timedelta type.

**Parameters**

- `arg` [str, timedelta, list-like or Series] The data to be converted to timedelta.

  Deprecated since version 1.2: Strings with units `M`, `Y` and `y` do not represent unambiguous timedelta values and will be removed in a future version.
**unit**  [str, optional] Denotes the unit of the arg for numeric *arg*. Defaults to "ns".

Possible values:

- ‘W’
- ‘D’ / ‘days’ / ‘day’
- ‘hours’ / ‘hour’ / ‘hr’ / ‘h’
- ‘m’ / ‘minute’ / ‘min’ / ‘minutes’ / ‘T’
- ‘S’ / ‘seconds’ / ‘sec’ / ‘second’
- ‘ms’ / ‘milliseconds’ / ‘millisecond’ / ‘milli’ / ‘millis’ / ‘L’
- ‘us’ / ‘microseconds’ / ‘microsecond’ / ‘micro’ / ‘micros’ / ‘U’
- ‘ns’ / ‘nanoseconds’ / ‘nano’ / ‘nanos’ / ‘nanosecond’ / ‘N’

Changed in version 1.1.0: Must not be specified when *arg* context strings and **errors**= "raise".

**errors**  [{‘ignore’, ‘raise’, ‘coerce’}, default ‘raise’]

- If ‘raise’, then invalid parsing will raise an exception.
- If ‘coerce’, then invalid parsing will be set as NaT.
- If ‘ignore’, then invalid parsing will return the input.

**Returns**

*timedelta* If parsing succeeded. Return type depends on input:

- list-like: TimedeltaIndex of timedelta64 dtype
- Series: Series of timedelta64 dtype
- scalar: Timedelta

**See also:**

- **DataFrame.astype** Cast argument to a specified dtype.
- **to_datetime** Convert argument to datetime.
- **convert_dtypes** Convert dtypes.

**Notes**

If the precision is higher than nanoseconds, the precision of the duration is truncated to nanoseconds for string inputs.
Examples

Parsing a single string to a Timedelta:

```python
>>> pd.to_timedelta('1 days 06:05:01.00003')
Timedelta('1 days 06:05:01.000030')

>>> pd.to_timedelta('15.5us')
Timedelta('0 days 00:00:00.000015500')
```

Parsing a list or array of strings:

```python
>>> pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015500', NaT],
dtype='timedelta64[ns]', freq=None)
```

Converting numbers by specifying the `unit` keyword argument:

```python
>>> pd.to_timedelta(np.arange(5), unit='s')
TimedeltaIndex(['0 days 00:00:00', '0 days 00:00:01', '0 days 00:00:02',
                '0 days 00:00:03', '0 days 00:00:04'],
dtype='timedelta64[ns]', freq=None)

>>> pd.to_timedelta(np.arange(5), unit='d')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
dtype='timedelta64[ns]', freq=None)
```

**pandas.date_range**

```python
pandas.date_range(start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=NoDefault.no_default, inclusive=None, **kwargs)
```

Return a fixed frequency DatetimeIndex.

Returns the range of equally spaced time points (where the difference between any two adjacent points is specified by the given frequency) such that they all satisfy `start <= x <= end`, where the first one and the last one are, resp., the first and last time points in that range that fall on the boundary of `freq` (if given as a frequency string) or that are valid for `freq` (if given as a pandas.tseries.offsets.DateOffset). (If exactly one of `start`, `end`, or `freq` is not specified, this missing parameter can be computed given `periods`, the number of timesteps in the range. See the note below.)

**Parameters**

- **start** [str or datetime-like, optional] Left bound for generating dates.
- **end** [str or datetime-like, optional] Right bound for generating dates.
- **periods** [int, optional] Number of periods to generate.
- **freq** [str or DateOffset, default ‘D’] Frequency strings can have multiples, e.g. ‘5H’. See here for a list of frequency aliases.
- **tz** [str or tzinfo, optional] Time zone name for returning localized DatetimeIndex, for example ‘Asia/Hong_Kong’. By default, the resulting DatetimeIndex is timezone-naive.
- **normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.
- **name** [str, default None] Name of the resulting DatetimeIndex.
closed  [{None, ‘left’, ‘right’}, optional]  Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None, the default).

Deprecation since version 1.4.0: Argument closed has been deprecated to standardize boundary inputs. Use inclusive instead, to set each bound as closed or open.

inclusive  [{‘both’, ‘neither’, ‘left’, ‘right’}, default “both”] Include boundaries; Whether to set each bound as closed or open.

New in version 1.4.0.

**kwargs  For compatibility. Has no effect on the result.

Returns

rng  [DatetimeIndex]

See also:

DatetimeIndex  An immutable container for datetimes.
timedelta_range  Return a fixed frequency TimedeltaIndex.
period_range  Return a fixed frequency PeriodIndex.
interval_range  Return a fixed frequency IntervalIndex.

Notes

Of the four parameters start, end, periods, and freq, exactly three must be specified. If freq is omitted, the resulting DatetimeIndex will have periods linearly spaced elements between start and end (closed on both sides).

To learn more about the frequency strings, please see this link.

Examples

Specifying the values

The next four examples generate the same DatetimeIndex, but vary the combination of start, end and periods.

Specify start and end, with the default daily frequency.

```python
>>> pd.date_range(start='1/1/2018', end='1/08/2018')
              dtype='datetime64[ns]', freq='D')
```

Specify start and periods, the number of periods (days).

```python
>>> pd.date_range(start='1/1/2018', periods=8)
              dtype='datetime64[ns]', freq='D')
```

Specify end and periods, the number of periods (days).
>>> pd.date_range(end='1/1/2018', periods=8)
DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
               '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
               dtype='datetime64[ns]', freq='D')

Specify start, end, and periods; the frequency is generated automatically (linearly spaced).

>>> pd.date_range(start='2018-04-24', end='2018-04-27', periods=3)
DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
               '2018-04-27 00:00:00'],
               dtype='datetime64[ns]', freq=None)

Other Parameters

Changed the freq (frequency) to 'M' (month end frequency).

>>> pd.date_range(start='1/1/2018', periods=5, freq='M')
               '2018-05-31'],
               dtype='datetime64[ns]', freq='M')

Multiples are allowed

>>> pd.date_range(start='1/1/2018', periods=5, freq='3M')
               '2019-01-31'],
               dtype='datetime64[ns]', freq='3M')

freq can also be specified as an Offset object.

>>> pd.date_range(start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3))
               '2019-01-31'],
               dtype='datetime64[ns]', freq='3M')

Specify tz to set the timezone.

>>> pd.date_range(start='1/1/2018', periods=5, tz='Asia/Tokyo')
DatetimeIndex(['2018-01-01 00:00:00+09:00', '2018-01-02 00:00:00+09:00',
               '2018-01-03 00:00:00+09:00', '2018-01-04 00:00:00+09:00',
               '2018-01-05 00:00:00+09:00'],
               dtype='datetime64[ns, Asia/Tokyo]', freq='D')

inclusive controls whether to include start and end that are on the boundary. The default, “both”, includes boundary points on either end.

>>> pd.date_range(start='2017-01-01', end='2017-01-04', inclusive="both")
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
               dtype='datetime64[ns]', freq='D')

Use inclusive='left' to exclude end if it falls on the boundary.

>>> pd.date_range(start='2017-01-01', end='2017-01-04', inclusive='left')
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'],
               dtype='datetime64[ns]', freq='D')
Use `inclusive='right'` to exclude `start` if it falls on the boundary, and similarly `inclusive='neither'` will exclude both `start` and `end`.

```python
>>> pd.date_range(start='2017-01-01', end='2017-01-04', inclusive='right')
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

### pandas.bdate_range

`pandas.bdate_range(start=None, end=None, periods=None, freq='B', tz=None, normalize=True, name=None, weekmask=None, holidays=None, closed=NoDefault.no_default, inclusive=None, **kwargs)`

Return a fixed frequency DatetimeIndex, with business day as the default frequency.

**Parameters**

- **start** [str or datetime-like, default None] Left bound for generating dates.
- **end** [str or datetime-like, default None] Right bound for generating dates.
- **periods** [int, default None] Number of periods to generate.
- **freq** [str or DateOffset, default ‘B’ (business daily)] Frequency strings can have multiples, e.g. ‘5H’.
- **tz** [str or None] Time zone name for returning localized DatetimeIndex, for example Asia/Beijing.
- **normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.
- **name** [str, default None] Name of the resulting DatetimeIndex.
- **weekmask** [str or None, default None] Weekmask of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed. The default value None is equivalent to ‘Mon Tue Wed Thu Fri’.
- **holidays** [list-like or None, default None] Dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed.
- **closed** [str, default None] Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None).
  
  Deprecated since version 1.4.0: Argument `closed` has been deprecated to standardize boundary inputs. Use `inclusive` instead, to set each bound as closed or open.
- **inclusive** [(“both”, “neither”, “left”, “right”), default “both”] Include boundaries: Whether to set each bound as closed or open.
  
  New in version 1.4.0.
- ****kwargs For compatibility. Has no effect on the result.

**Returns**

`DatetimeIndex`
Notes

Of the four parameters: `start`, `end`, `periods`, and `freq`, exactly three must be specified. Specifying `freq` is a requirement for `bdate_range`. Use `date_range` if specifying `freq` is not desired.

To learn more about the frequency strings, please see this link.

Examples

Note how the two weekend days are skipped in the result.

```python
>>> pd.bdate_range(start='1/1/2018', end='1/08/2018')
dtype='datetime64[ns]', freq='B')
```

`pandas.period_range`

`pandas.period_range(start=None, end=None, periods=None, freq=None, name=None)`

Return a fixed frequency PeriodIndex.

The day (calendar) is the default frequency.

Parameters

- `start` [str or period-like, default None] Left bound for generating periods.
- `end` [str or period-like, default None] Right bound for generating periods.
- `periods` [int, default None] Number of periods to generate.
- `freq` [str or DateOffset, optional] Frequency alias. By default the freq is taken from `start` or `end` if those are Period objects. Otherwise, the default is "D" for daily frequency.
- `name` [str, default None] Name of the resulting PeriodIndex.

Returns

PeriodIndex

Notes

Of the three parameters: `start`, `end`, and `periods`, exactly two must be specified.

To learn more about the frequency strings, please see this link.

Examples

```python
>>> pd.period_range(start='2017-01-01', end='2018-01-01', freq='M')
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', '2017-05', '2017-06',
             '2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
             '2018-01'],
dtype='period[M]')
```

If `start` or `end` are Period objects, they will be used as anchor endpoints for a PeriodIndex with frequency matching that of the period_range constructor.
```python
>>> pd.period_range(start=pd.Period('2017Q1', freq='Q'),
...                   end=pd.Period('2017Q2', freq='Q'), freq='M')
PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'],
            dtype='period[M]')
```

### pandas.timedelta_range

**pandas.timedelta_range**

`start=None, end=None, periods=None, freq=None, name=None, closed=None`

Return a fixed frequency TimedeltaIndex, with day as the default frequency.

**Parameters**

- `start` [str or timedelta-like, default None] Left bound for generating timedeltas.
- `end` [str or timedelta-like, default None] Right bound for generating timedeltas.
- `periods` [int, default None] Number of periods to generate.
- `freq` [str or DateOffset, default ‘D’] Frequency strings can have multiples, e.g. ‘5H’.
- `name` [str, default None] Name of the resulting TimedeltaIndex.
- `closed` [str, default None] Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None).

**Returns**

TimedeltaIndex

**Notes**

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting TimedeltaIndex will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see this link.

**Examples**

```python
>>> pd.timedelta_range(start='1 day', periods=4)
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The `closed` parameter specifies which endpoint is included. The default behavior is to include both endpoints.

```python
>>> pd.timedelta_range(start='1 day', periods=4, closed='right')
TimedeltaIndex(['2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The `freq` parameter specifies the frequency of the TimedeltaIndex. Only fixed frequencies can be passed, non-fixed frequencies such as ‘M’ (month end) will raise.
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> pd.timedelta_range(start='1 day', end='2 days', freq='6H')
TimedeltaIndex(['1 days 00:00:00', '1 days 06:00:00', '1 days 12:00:00', '1 days 18:00:00', '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='6H')

Specify start, end, and periods; the frequency is generated automatically (linearly spaced).

>>> pd.timedelta_range(start='1 day', end='5 days', periods=4)
TimedeltaIndex(['1 days 00:00:00', '2 days 08:00:00', '3 days 16:00:00', '5 days 00:00:00'],
                dtype='timedelta64[ns]', freq=None)

pandas.infer_freq

pandas.infer_freq(index, warn=True)

Infer the most likely frequency given the input index. If the frequency is uncertain, a warning will be printed.

Parameters

- **index** [DatetimeIndex or TimedeltaIndex] If passed a Series will use the values of the series (NOT THE INDEX).
- **warn** [bool, default True]

Returns

- **str or None** None if no discernible frequency.

Raises

- **TypeError** If the index is not datetime-like.
- **ValueError** If there are fewer than three values.

Examples

>>> idx = pd.date_range(start='2020/12/01', end='2020/12/30', periods=30)
>>> pd.infer_freq(idx)
'D'

3.2.5 Top-level dealing with Interval data

interval_range([start, end, periods, freq, ...]) Return a fixed frequency IntervalIndex.
pandas.interval_range

pandas.interval_range(start=None, end=None, periods=None, freq=None, name=None, closed='right')

Return a fixed frequency IntervalIndex.

Parameters

- **start** [numeric or datetime-like, default None] Left bound for generating intervals.
- **end** [numeric or datetime-like, default None] Right bound for generating intervals.
- **periods** [int, default None] Number of periods to generate.
- **freq** [numeric, str, or DateOffset, default None] The length of each interval. Must be consistent with the type of start and end, e.g. 2 for numeric, or ‘5H’ for datetime-like. Default is 1 for numeric and ‘D’ for datetime-like.
- **name** [str, default None] Name of the resulting IntervalIndex.
- **closed** [['left', 'right', 'both', 'neither'], default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

IntervalIndex

See also:

IntervalIndex  An Index of intervals that are all closed on the same side.

Notes

Of the four parameters start, end, periods, and freq, exactly three must be specified. If freq is omitted, the resulting IntervalIndex will have periods linearly spaced elements between start and end, inclusively.

To learn more about datetime-like frequency strings, please see this link.

Examples

Numeric start and end is supported.

```python
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
dtype='interval[int64, right]')
```

Additionally, datetime-like input is also supported.

```python
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
                  end=pd.Timestamp('2017-01-04'))
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03],
               (2017-01-03, 2017-01-04]],
dtype='interval[datetime64[ns], right]')
```

The freq parameter specifies the frequency between the left and right endpoints of the individual intervals within the IntervalIndex. For numeric start and end, the frequency must also be numeric.
```python
>>> pd.interval_range(start=0, periods=4, freq=1.5)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
dtype='interval[float64, right]')
```

Similarly, for datetime-like `start` and `end`, the frequency must be convertible to a DateOffset.

```python
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
...                    periods=3, freq='MS')
IntervalIndex([(2017-01-01, 2017-02-01], (2017-02-01, 2017-03-01],
               (2017-03-01, 2017-04-01]],
dtype='interval[datetime64[ns], right]')
```

Specify `start`, `end`, and `periods`; the frequency is generated automatically (linearly spaced).

```python
>>> pd.interval_range(start=0, end=6, periods=4)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
dtype='interval[float64, right]')
```

The `closed` parameter specifies which endpoints of the individual intervals within the `IntervalIndex` are closed.

```python
>>> pd.interval_range(end=5, periods=4, closed='both')
IntervalIndex([[1, 2], [2, 3], [3, 4], [4, 5]],
dtype='interval[int64, both]')
```

### 3.2.6 Top-level evaluation

**eval**(expr[, parser, engine, truediv, ...]) Evaluate a Python expression as a string using various backends.

**pandas.eval**

**pandas.eval**(expr, parser='pandas', engine=None, truediv=NoDefault.no_default, local_dict=None, global_dict=None, resolvers=(), level=0, target=None, inplace=False)

Evaluate a Python expression as a string using various backends.

The following arithmetic operations are supported: `+`, `-`, `*`, `/`, `**`, `%`, `//` (python engine only) along with the following boolean operations: `|` (or), `&` (and), and `~` (not). Additionally, the "pandas" parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators. *Series* and *DataFrame* objects are supported and behave as they would with plain ol' Python evaluation.

**Parameters**

- **expr** [str] The expression to evaluate. This string cannot contain any Python statements, only Python expressions.
- **parser** ["pandas", "python"], default "pandas" The parser to use to construct the syntax tree from the expression. The default of "pandas" parses code slightly different than standard Python. Alternatively, you can parse an expression using the "python" parser to retain strict Python semantics. See the *enhancing performance* documentation for more details.
- **engine** ["python", "numexpr"], default "numexpr" The engine used to evaluate the expression. Supported engines are
None: tries to use `numexpr`, falls back to `python`.

'numexpr': This default engine evaluates pandas objects using `numexpr` for large speed ups in complex expressions with large frames.

'python': Performs operations as if you had `eval'd in top` level `python`. This engine is generally not that useful.

More backends may be available in the future.

**truediv** [bool, optional] Whether to use true division, like in Python >= 3.

Deprecated since version 1.0.0.

**local_dict** [dict or None, optional] A dictionary of local variables, taken from `locals()` by default.

**global_dict** [dict or None, optional] A dictionary of global variables, taken from `globals()` by default.

**resolvers** [list of dict-like or None, optional] A list of objects implementing the `__getitem__` special method that you can use to inject an additional collection of namespaces to use for variable lookup. For example, this is used in the `query()` method to inject the `DataFrame.index` and `DataFrame.columns` variables that refer to their respective `DataFrame` instance attributes.

**level** [int, optional] The number of prior stack frames to traverse and add to the current scope. Most users will not need to change this parameter.

**target** [object, optional, default None] This is the target object for assignment. It is used when there is variable assignment in the expression. If so, then `target` must support item assignment with string keys, and if a copy is being returned, it must also support `.copy()`.

**inplace** [bool, default False] If `target` is provided, and the expression mutates `target`, whether to modify `target` inplace. Otherwise, return a copy of `target` with the mutation.

**Returns**

`ndarray`, `numeric scalar`, `DataFrame`, `Series`, or `None` The completion value of evaluating the given code or `None` if `inplace=True`.

**Raises**

**ValueError** There are many instances where such an error can be raised:

- `target=None`, but the expression is multiline.

- The expression is multiline, but not all them have item assignment. An example of such an arrangement is this:

```
a = b + 1  a + 2
```

Here, there are expressions on different lines, making it multiline, but the last line has no variable assigned to the output of `a + 2`.

- `inplace=True`, but the expression is missing item assignment.

- Item assignment is provided, but the `target` does not support string item assignment.

- Item assignment is provided and `inplace=False`, but the `target` does not support the `.copy()` method.

**See also:**

`DataFrame.query` Evaluates a boolean expression to query the columns of a frame.

3.2. General functions
**DataFrame.eval** Evaluate a string describing operations on DataFrame columns.

**Notes**

The dtype of any objects involved in an arithmetic % operation are recursively cast to float64. See the *enhancing performance* documentation for more details.

**Examples**

```python
def = pd.DataFrame({"animal": ["dog", "pig"], "age": [10, 20]})
def
   animal  age
   0   dog   10
   1   pig   20
```

We can add a new column using `pd.eval`:

```python
>>> pd.eval("double_age = df.age * 2", target=df)
df
   animal  age  double_age
   0   dog   10       20
   1   pig   20       40
```

### 3.2.7 Hashing

**util.hash_array**

Given a 1d array, return an array of deterministic integers.

**util.hash_pandas_object**

Return a data hash of the Index/Series/DataFrame.

**pandas.util.hash_array**

Given a 1d array, return an array of deterministic integers.

**Parameters**

- **vals** [ndarray or ExtensionArray]
- **encoding** [str, default 'utf8'] Encoding for data & key when strings.
- **hash_key** [str, default _default_hash_key] Hash_key for string key to encode.
- **categorize** [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

**Returns**

- **ndarray[<class 'numpy.uint64'>, ndim=1]** Hashed values, same length as the vals.
pandas.util.hash_pandas_object

Return a data hash of the Index/Series/DataFrame.

Parameters
- `obj` [Index, Series, or DataFrame]
- `index` [bool, default True] Include the index in the hash (if Series/DataFrame).
- `encoding` [str, default ‘utf8’] Encoding for data & key when strings.
- `hash_key` [str, default _default_hash_key] Hash_key for string key to encode.
- `categorize` [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

Returns
- Series of uint64, same length as the object

3.2.8 Testing

```python
test([extra_args])
```
Run the pandas test suite using pytest.

pandas.test

```python
pandas.test(extra_args=None)
```
Run the pandas test suite using pytest.

3.3 Series

3.3.1 Constructor

```python
Series([data, index, dtype, name, copy, ...])
```
One-dimensional ndarray with axis labels (including time series).

pandas.Series

```python
class pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
```
One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, *, **) align values based on their associated index values— they need not be the same length. The result index will be the sorted union of the two indexes.

Parameters
**data** [array-like, iterable, dict, or scalar value] Contains data stored in Series. If data is a dict, argument order is maintained.

**index** [array-like or Index (1d)] Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, ..., n) if not provided. If data is dict-like and index is None, then the keys in the data are used as the index. If the index is not None, the resulting Series is reindexed with the index values.

**dtype** [str, numpy.dtype, or ExtensionDtype, optional] Data type for the output Series. If not specified, this will be inferred from *data*. See the user guide for more usages.

**name** [str, optional] The name to give to the Series.

**copy** [bool, default False] Copy input data. Only affects Series or 1d ndarray input. See examples.

### Examples

**Constructing Series from a dictionary with an Index specified**

```python
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> ser = pd.Series(data=d, index=['a', 'b', 'c'])
>>> ser
a 1  
b 2  
c 3  
dtype: int64
```

The keys of the dictionary match with the Index values, hence the Index values have no effect.

```python
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> ser = pd.Series(data=d, index=['x', 'y', 'z'])
>>> ser
x NaN 
y NaN 
z NaN 
dtype: float64
```

Note that the Index is first build with the keys from the dictionary. After this the Series is reindexed with the given Index values, hence we get all NaN as a result.

**Constructing Series from a list with copy=False**

```python
>>> r = [1, 2]
>>> ser = pd.Series(r, copy=False)
>>> ser.iloc[0] = 999
>>> r
[1, 2]
>>> ser
0 999
1 2

dtype: int64
```

Due to input data type the Series has a copy of the original data even though copy=False, so the data is unchanged.

**Constructing Series from a 1d ndarray with copy=False**

```python
```
Due to input data type the Series has a view on the original data, so the data is changed as well.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Return the transpose, which is by definition self.</td>
</tr>
<tr>
<td><code>array</code></td>
<td>The ExtensionArray of the data backing this Series or Index.</td>
</tr>
<tr>
<td><code>at</code></td>
<td>Access a single value for a row/column label pair.</td>
</tr>
<tr>
<td><code>attrs</code></td>
<td>Dictionary of global attributes of this dataset.</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>Return a list of the row axis labels.</td>
</tr>
<tr>
<td><code>dtypes</code></td>
<td>Return the dtypes object of the underlying data.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Get the properties associated with this pandas object.</td>
</tr>
<tr>
<td><code>hasnans</code></td>
<td>Return True if there are any NaNs.</td>
</tr>
<tr>
<td><code>iat</code></td>
<td>Access a single value for a row/column pair by integer position.</td>
</tr>
<tr>
<td><code>iloc</code></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The index (axis labels) of the Series.</td>
</tr>
<tr>
<td><code>is_monotonic</code></td>
<td>Return boolean if values in the object are monotonic_increasing.</td>
</tr>
<tr>
<td><code>is_monotonic_decreasing</code></td>
<td>Return boolean if values in the object are monotonic_decreasing.</td>
</tr>
<tr>
<td><code>is_monotonic_increasing</code></td>
<td>Alias for is_monotonic.</td>
</tr>
<tr>
<td><code>is_unique</code></td>
<td>Return boolean if values in the object are unique.</td>
</tr>
<tr>
<td><code>loc</code></td>
<td>Access a group of rows and columns by label(s) or a boolean array.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Return the name of the Series.</td>
</tr>
<tr>
<td><code>nbytes</code></td>
<td>Return the number of bytes in the underlying data.</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td>Number of dimensions of the underlying data, by definition 1.</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>Return a tuple of the shape of the underlying data.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Return the number of elements in the underlying data.</td>
</tr>
<tr>
<td><code>values</code></td>
<td>Return Series as ndarray or ndarray-like depending on the dtypes.</td>
</tr>
</tbody>
</table>
pandas.Series.T

**property** Series.T
Return the transpose, which is by definition self.

pandas.Series.array

**property** Series.array
The ExtensionArray of the data backing this Series or Index.

**Returns**

ExtensionArray An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around numpy.ndarray.

.array differs .values which may require converting the data to a different form.

**See also:**

Index.to_numpy Similar method that always returns a NumPy array.
Series.to_numpy Similar method that always returns a NumPy array.

**Notes**

This table lays out the different array types for each extension dtype within pandas.

<table>
<thead>
<tr>
<th>dtype</th>
<th>array type</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>Categorical</td>
</tr>
<tr>
<td>period</td>
<td>PeriodArray</td>
</tr>
<tr>
<td>interval</td>
<td>IntervalArray</td>
</tr>
<tr>
<td>IntegerNA</td>
<td>IntegerArray</td>
</tr>
<tr>
<td>string</td>
<td>StringArray</td>
</tr>
<tr>
<td>boolean</td>
<td>BooleanArray</td>
</tr>
<tr>
<td>datetime64[ns, tz]</td>
<td>DatetimeArray</td>
</tr>
</tbody>
</table>

For any 3rd-party extension types, the array type will be an ExtensionArray.

For all remaining dtypes .array will be a arrays.NumpyExtensionArray wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use Series.to_numpy() instead.

**Examples**

For regular NumPy types like int, and float, a PandasArray is returned.

```python
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned.
```python
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
['a', 'b', 'a']
Categories (2, object): ['a', 'b']
```

**pandas.Series.at**

**property Series.at**

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

**Raises**

`KeyError` If ‘label’ does not exist in DataFrame.

**See also:**

`DataFrame.iat` Access a single value for a row/column pair by integer position.

`DataFrame.loc` Access a group of rows and columns by label(s).

`Series.at` Access a single value using a label.

**Examples**

```python
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
   A  B  C
4  0  2  3
5  0  4  1
6 10 20 30
```

Get value at specified row/column pair

```python
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```python
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```python
>>> df.loc[5].at['B']
4
```
**pandas.Series.attrs**

**property** Series.attrs

Dictionary of global attributes of this dataset.

**Warning:** attrs is experimental and may change without warning.

See also:

*DataFrame.flags* Global flags applying to this object.

**pandas.Series.axes**

**property** Series.axes

Return a list of the row axis labels.

**pandas.Series.dtype**

**property** Series.dtype

Return the dtype object of the underlying data.

**pandas.Series.dtypes**

**property** Series.dtypes

Return the dtype object of the underlying data.

**pandas.Series.flags**

**property** Series.flags

Get the properties associated with this pandas object.

The available flags are

- *Flags.allows_duplicate_labels*

See also:

*Flags* Flags that apply to pandas objects.

*DataFrame.attrs* Global metadata applying to this dataset.
Notes

“Flags” differ from “metadata”. Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```python
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags
<Flags(allows_duplicate_labels=True)>
```

Flags can be get or set using 

```python
>>> df.flags.allows_duplicate_labels
True
>>> df.flags.allows_duplicate_labels = False
```

Or by slicing with a key

```python
>>> df.flags["allows_duplicate_labels"]
False
>>> df.flags["allows_duplicate_labels"] = True
```

```python
pandas.Series.hasnans
```

**property** `Series.hasnans`

Return True if there are any NaNs.

Enables various performance speedups.

```python
pandas.Series.iat
```

**property** `Series.iat`

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

- `IndexError` When integer position is out of bounds.

See also:

- `DataFrame.at` Access a single value for a row/column label pair.
- `DataFrame.loc` Access a group of rows and columns by label(s).
- `DataFrame.iloc` Access a group of rows and columns by integer position(s).
Examples

```python
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                   columns=['A', 'B', 'C'])

>>> df
       A  B  C
0     0  2  3
1     0  4  1
2    10 20 30
```

Get value at specified row/column pair

```python
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```python
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```python
>>> df.loc[0].iat[1]
2
```

**pandas.Series.iloc**

**property** *Series.iloc*

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don’t have a reference to the calling object, but would like to base your selection on some value.

.iloc will raise *IndexError* if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

See more at *Selection by Position*.

**See also:**

<Dataframe.iat> Fast integer location scalar accessor.

<Dataframe.loc> Purely label-location based indexer for selection by label.
Series.iloc  Purely integer-location based indexing for selection by position.

Examples

```python
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
        {'a': 100, 'b': 200, 'c': 300, 'd': 400},
        {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000}]
>>> df = pd.DataFrame(mydict)
>>> df
  a  b  c  d
0  1  2  3  4
1 100 200 300 400
2 1000 2000 3000 4000

Indexing just the rows

With a scalar integer.

```python
>>> df.iloc[0]
    a    b    c    d
0   1    2    3    4
```

With a list of integers.

```python
>>> df.iloc[[0]]
    a    b    c    d
0   1    2    3    4
```

With a slice object.

```python
>>> df.iloc[:3]
   a  b  c  d
0  1  2  3  4
1 100 200 300 400
2 1000 2000 3000 4000
```

With a boolean mask the same length as the index.

```python
>>> df.iloc[[True, False, True]]
   a  b  c  d
0  1  2  3  4
2 1000 2000 3000 4000
```
With a callable, useful in method chains. The x passed to the lambda is the DataFrame being sliced. This selects the rows whose index label even.

```python
>>> df.iloc[lambda x: x.index % 2 == 0]
a  b  c   d
0  1  2   3   4
2 1000 2000 3000 4000
```

**Indexing both axes**

You can mix the indexer types for the index and columns. Use : to select the entire axis.

With scalar integers.

```python
>>> df.iloc[0, 1]
2
```

With lists of integers.

```python
>>> df.iloc[[0, 2], [1, 3]]
b  d
0  2  4
2 2000 4000
```

With `slice` objects.

```python
>>> df.iloc[1:3, 0:3]
a  b  c
1 100 200 300
2 1000 2000 3000
```

With a boolean array whose length matches the columns.

```python
>>> df.iloc[:, [True, False, True, False]]
a  c
0  1  3
1 100 300
2 1000 3000
```

With a callable function that expects the Series or DataFrame.

```python
>>> df.iloc[:, lambda df: [0, 2]]
a  c
0  1  3
1 100 300
2 1000 3000
```
pandas.Series.index

Series.index
The index (axis labels) of the Series.

pandas.Series.is_monotonic

property Series.is_monotonic
Return boolean if values in the object are monotonic_increasing.

Returns
bool

pandas.Series.is_monotonic_decreasing

property Series.is_monotonic_decreasing
Return boolean if values in the object are monotonic_decreasing.

Returns
bool

pandas.Series.is_monotonic_increasing

property Series.is_monotonic_increasing
Alias for is_monotonic.

pandas.Series.is_unique

property Series.is_unique
Return boolean if values in the object are unique.

Returns
bool

pandas.Series.loc

property Series.loc
Access a group of rows and columns by label(s) or a boolean array.
.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

• A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).

• A list or array of labels, e.g. ['a', 'b', 'c'].

3.3. Series
A slice object with labels, e.g. 'a':'f'.

**Warning:** Note that contrary to usual python slices, both the start and the stop are included.

- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- An alignable boolean Series. The index of the key will be aligned before masking.
- An alignable Index. The Index of the returned selection will be the input.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

See more at *Selection by Label.*

**Raises**

- **KeyError** If any items are not found.
- **IndexingError** If an indexed key is passed and its index is unalignable to the frame index.

**See also:**

- `DataFrame.at` Access a single value for a row/column label pair.
- `DataFrame.iloc` Access group of rows and columns by integer position(s).
- `DataFrame.xs` Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.
- `Series.loc` Access group of values using labels.

**Examples**

**Getting values**

```python
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
                  index=['cobra', 'viper', 'sidewinder'],
                  columns=['max_speed', 'shield'])
>>> df
                        max_speed  shield
    cobra          1          2
    viper          4          5
    sidewinder     7          8
```

Single label. Note this returns the row as a Series.

```python
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using [[]] returns a DataFrame.

```python
>>> df.loc[['viper', 'sidewinder']]
                        max_speed  shield
    cobra          1          2
    viper          4          5
    sidewinder     7          8
```
Single label for row and column

```python
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```python
>>> df.loc['cobra':'viper', 'max_speed']
cobra 1
viper 4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```python
>>> df.loc[[False, False, True]]
      max_speed  shield
sidewinder          7     8
```

Alignable boolean Series:

```python
>>> df.loc[pd.Series([False, True, False],
...                   index=['viper', 'sidewinder', 'cobra'])]
      max_speed  shield
sidewinder          7     8
```

Index (same behavior as `df.reindex`)

```python
>>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
      max_speed  shield
foo             
  cobra          1     2
  viper          4     5
```

Conditional that returns a boolean Series

```python
>>> df.loc[df['shield'] > 6]
      max_speed  shield
sidewinder          7     8
```

Conditional that returns a boolean Series with column labels specified

```python
>>> df.loc[df['shield'] > 6, ['max_speed']]  
      max_speed
sidewinder          7
```

Callable that returns a boolean Series

```python
>>> df.loc[lambda df: df['shield'] == 8]
      max_speed  shield
sidewinder          7     8
```

Setting values
Set value for all items matching the list of labels
Set value for an entire row:

```python
>>> df.loc['cobra'] = 10
>>> df
   max_speed shield
cobra    10    10
viper     4     50
sidewinder 7     50
```

Set value for an entire column:

```python
>>> df.loc[:, 'max_speed'] = 30
>>> df
   max_speed shield
cobra    30    10
viper     4     50
sidewinder 7     50
```

Set value for rows matching callable condition:

```python
>>> df.loc[df['shield'] > 35] = 0
>>> df
   max_speed shield
cobra     30     10
viper      0      0
sidewinder 0      0
```

Getting values on a DataFrame with an index that has integer labels:

Another example using integers for the index:

```python
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
                    index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed shield
   7         1     2
   8         4     5
   9         7     8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included:

```python
>>> df.loc[7:9]
   max_speed shield
   7         1     2
   8         4     5
   9         7     8
```
Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```python
>>> tuples = [  
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),  
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),  
...     ('viper', 'mark ii'), ('viper', 'mark iii')  
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],  
... [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df

max_speed  shield
cobra mark i  12  2
mark ii    0   4
sidewinder mark i  10  20
mark ii    1   4
viper mark ii    7  1
mark iii   16  36
```

Single label. Note this returns a DataFrame with a single index.

```python
>>> df.loc['cobra']

max_speed  shield
mark i     12  2
mark ii    0   4
```

Single index tuple. Note this returns a Series.

```python
>>> df.loc[('cobra', 'mark ii')]

max_speed   0
shield      4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```python
>>> df.loc['cobra', 'mark i']

max_speed  12
shield     2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using [[]] returns a DataFrame.

```python
>>> df.loc[['cobra', 'mark ii']]  

max_speed  shield
cobra mark ii  0   4
```

Single tuple for the index with a single label for the column

```python
>>> df.loc[['cobra', 'mark i'], 'shield']

2
```

Slice from index tuple to single label
```python
>>> df.loc[['cobra', 'mark i']:, 'viper']
      max_speed  shield
  cobra      mark i  12     2
          mark ii   0     4
sidewinder  mark i  10    20
          mark ii   1     4
viper      mark ii   7     1
          mark iii  16    36
```

Slice from index tuple to index tuple

```python
>>> df.loc[['cobra', 'mark i']:, ('viper', 'mark ii')]
      max_speed  shield
  cobra      mark i  12     2
          mark ii   0     4
sidewinder  mark i  10    20
          mark ii   1     4
viper      mark ii   7     1
```

### pandas.Series.name

**property Series.name**

Return the name of the Series.

The name of a Series becomes its index or column name if it is used to form a DataFrame. It is also used whenever displaying the Series using the interpreter.

**Returns**

- **label (hashable object)**: The name of the Series, also the column name if part of a DataFrame.

**See also:**

- **Series.rename**: Sets the Series name when given a scalar input.
- **Index.name**: Corresponding Index property.

**Examples**

The Series name can be set initially when calling the constructor.

```python
>>> s = pd.Series([1, 2, 3], dtype=np.int64, name='Numbers')
>>> s
0  1
1  2
2  3
Name: Numbers, dtype: int64
>>> s.name = "Integers"
>>> s
0  1
1  2
2  3
Name: Integers, dtype: int64
```
The name of a Series within a DataFrame is its column name.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4], [5, 6]],
...                    columns=["Odd Numbers", "Even Numbers"]
>>> df
   Odd Numbers   Even Numbers
0       1         2
1       3         4
2       5         6
>>> df["Even Numbers"].name
'Even Numbers'
```

**pandas.Series.nbytes**

**property** pandas.Series.nbytes

Return the number of bytes in the underlying data.

**pandas.Series.ndim**

**property** pandas.Series.ndim

Number of dimensions of the underlying data, by definition 1.

**pandas.Series.shape**

**property** pandas.Series.shape

Return a tuple of the shape of the underlying data.

**pandas.Series.size**

**property** pandas.Series.size

Return the number of elements in the underlying data.

**pandas.Series.values**

**property** pandas.Series.values

Return Series as ndarray or ndarray-like depending on the dtype.

**Warning:** We recommend using `Series.array` or `Series.to_numpy()`, depending on whether you need a reference to the underlying data or a NumPy array.

Returns

- `numpy.ndarray` or `ndarray-like`

See also:

- `Series.array` Reference to the underlying data.
Series.to_numpy A NumPy array representing the underlying data.

Examples

```python
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```python
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```python
>>> pd.Series(list('aabc')).astype('category').values
['a', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Timezone aware datetime data is converted to UTC:

```python
>>> pd.Series(pd.date_range('20130101', periods=3, tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype=datet
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Return a Series/DataFrame with absolute numeric value of each element.</td>
</tr>
<tr>
<td>add(other[, level, fill_value, axis])</td>
<td>Return Addition of series and other, element-wise (binary operator add).</td>
</tr>
<tr>
<td>add_prefix(prefix)</td>
<td>Prefix labels with string prefix.</td>
</tr>
<tr>
<td>add_suffix(suffix)</td>
<td>Suffix labels with string suffix.</td>
</tr>
<tr>
<td>agg(func, axis)</td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td>aggregate(func, axis)</td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td>align(other[, join, axis, level, copy, ...])</td>
<td>Align two objects on their axes with the specified join method.</td>
</tr>
<tr>
<td>all([axis, bool_only, skipna, level])</td>
<td>Return whether all elements are True, potentially over an axis.</td>
</tr>
<tr>
<td>any([axis, bool_only, skipna, level])</td>
<td>Return whether any element is True, potentially over an axis.</td>
</tr>
<tr>
<td>append(to_append[, ignore_index, ...])</td>
<td>(DEPRECATED) Concatenate two or more Series.</td>
</tr>
<tr>
<td>apply(func[, convert_dtype, args])</td>
<td>Invoke function on values of Series.</td>
</tr>
<tr>
<td>argmax([axis, skipna])</td>
<td>Return int position of the largest value in the Series.</td>
</tr>
<tr>
<td>argmin([axis, skipna])</td>
<td>Return int position of the smallest value in the Series.</td>
</tr>
<tr>
<td>argsort([axis, kind, order])</td>
<td>Return the integer indices that would sort the Series values.</td>
</tr>
<tr>
<td>asfreq(freq[, method, how, normalize, ...])</td>
<td>Convert time series to specified frequency.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asof(where[, subset])</code></td>
<td>Return the last row(s) without any NaNs before <code>where</code>.</td>
</tr>
<tr>
<td><code>astype(dtype[, copy, errors])</code></td>
<td>Cast a pandas object to a specified dtype <code>dtype</code>.</td>
</tr>
<tr>
<td><code>at_time(time[, asof, axis])</code></td>
<td>Select values at particular time of day (e.g., 9:30AM).</td>
</tr>
<tr>
<td><code>autocorr(lag)</code></td>
<td>Compute the lag-N autocorrelation.</td>
</tr>
<tr>
<td><code>backfill([axis, inplace, limit, downcast])</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
</tr>
<tr>
<td><code>between(left, right[, inclusive])</code></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right.</td>
</tr>
<tr>
<td><code>between_time(start_time, end_time[...])</code></td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
</tr>
<tr>
<td><code>bfill([axis, inplace, limit, downcast])</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
</tr>
<tr>
<td><code>bool()</code></td>
<td>Return the bool of a single element Series or DataFrame.</td>
</tr>
<tr>
<td><code>cat</code></td>
<td>alias of <code>pandas.core.arrays.categorical.CategoricalAccessor</code></td>
</tr>
<tr>
<td><code>clip([lower, upper, axis, inplace])</code></td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><code>combine(other, func[, fill_value])</code></td>
<td>Combine the Series with a Series or scalar according to <code>func</code>.</td>
</tr>
<tr>
<td><code>combine_first(other)</code></td>
<td>Update null elements with value in the same location in 'other'.</td>
</tr>
<tr>
<td><code>compare(other[, align_axis, keep_shape, ...])</code></td>
<td>Compare to another Series and show the differences.</td>
</tr>
<tr>
<td><code>convert_dtypes([infer_objects, ...])</code></td>
<td>Convert columns to best possible dtypes using dtypes supporting pd.NA.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Make a copy of this object's indices and data.</td>
</tr>
<tr>
<td><code>corr(other[, method, min_periods])</code></td>
<td>Compute correlation with other Series, excluding missing values.</td>
</tr>
<tr>
<td><code>count([level])</code></td>
<td>Return number of non-NA/null observations in the Series.</td>
</tr>
<tr>
<td><code>cov(other[, min_periods, ddof])</code></td>
<td>Compute covariance with Series, excluding missing values.</td>
</tr>
<tr>
<td><code>cummax([axis, skipna])</code></td>
<td>Return cumulative maximum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><code>cummin([axis, skipna])</code></td>
<td>Return cumulative minimum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><code>cumprod([axis, skipna])</code></td>
<td>Return cumulative product over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><code>cumsum([axis, skipna])</code></td>
<td>Return cumulative sum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><code>describe([percentiles, include, exclude, ...])</code></td>
<td>Generate descriptive statistics.</td>
</tr>
<tr>
<td><code>diff([periods])</code></td>
<td>First discrete difference of element.</td>
</tr>
<tr>
<td><code>divide(other[, level, fill_value, axis])</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>divmod(other[, level, fill_value, axis])</code></td>
<td>Return Integer division and modulo of series and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>dot(other)</code></td>
<td>Compute the dot product between the Series and the columns of other.</td>
</tr>
<tr>
<td><code>drop([labels, axis, index, columns, level, ...])</code></td>
<td>Return Series with specified index labels removed.</td>
</tr>
</tbody>
</table>

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drop_duplicates()</code></td>
<td>Return Series with duplicate values removed.</td>
</tr>
<tr>
<td><code>droplevel()</code></td>
<td>Return Series/DataFrame with requested index / column level(s) removed.</td>
</tr>
<tr>
<td><code>dropna()</code></td>
<td>Return a new Series with missing values removed.</td>
</tr>
<tr>
<td><code>duplicated()</code></td>
<td>Indicate duplicate Series values.</td>
</tr>
<tr>
<td><code>eq()</code></td>
<td>Return Equal to of series and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
<tr>
<td><code>equals()</code></td>
<td>Test whether two objects contain the same elements.</td>
</tr>
<tr>
<td><code>ewm()</code></td>
<td>Provide exponentially weighted (EW) calculations.</td>
</tr>
<tr>
<td><code>expanding()</code></td>
<td>Provide expanding window calculations.</td>
</tr>
<tr>
<td><code>explode()</code></td>
<td>Transform each element of a list-like to a row.</td>
</tr>
<tr>
<td><code>factorize()</code></td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><code>ffill()</code></td>
<td>Synonym for DataFrame.fillna() with method='ffill'.</td>
</tr>
<tr>
<td><code>fillna()</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>filter()</code></td>
<td>Subset the dataframe rows or columns according to the specified index labels.</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>Select initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>first_valid_index()</code></td>
<td>Return index for first non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td><code>floor()</code></td>
<td>Return Integer division of series and other, element-wise (binary operator <code>floor</code>).</td>
</tr>
<tr>
<td><code>ge()</code></td>
<td>Return Greater than or equal to of series and other, element-wise (binary operator <code>ge</code>).</td>
</tr>
<tr>
<td><code>get()</code></td>
<td>Get item from object for given key (ex: DataFrame column).</td>
</tr>
<tr>
<td><code>groupby()</code></td>
<td>Group Series using a mapper or by a Series of columns.</td>
</tr>
<tr>
<td><code>gt()</code></td>
<td>Return Greater than of series and other, element-wise (binary operator <code>gt</code>).</td>
</tr>
<tr>
<td><code>head()</code></td>
<td>Return the first n rows.</td>
</tr>
<tr>
<td><code>hist()</code></td>
<td>Draw histogram of the input series using matplotlib.</td>
</tr>
<tr>
<td><code>idxmax()</code></td>
<td>Return the row label of the maximum value.</td>
</tr>
<tr>
<td><code>idxmin()</code></td>
<td>Return the row label of the minimum value.</td>
</tr>
<tr>
<td><code>infer_objects()</code></td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td><code>info()</code></td>
<td>Print a concise summary of a Series.</td>
</tr>
<tr>
<td><code>interpolate()</code></td>
<td>Fill NaN values using an interpolation method.</td>
</tr>
<tr>
<td><code>isin()</code></td>
<td>Whether elements in Series are contained in values.</td>
</tr>
<tr>
<td><code>isna()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>isnull()</code></td>
<td>Series.isnull is an alias for Series.isna.</td>
</tr>
<tr>
<td><code>item()</code></td>
<td>Return the first element of the underlying data as a Python scalar.</td>
</tr>
<tr>
<td><code>items()</code></td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td><code>iteritems()</code></td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Return alias for index.</td>
</tr>
<tr>
<td><code>kurt()</code></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td><code>kurtosis()</code></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>last(offset)</td>
<td>Select final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>last_valid_index()</td>
<td>Return index for last non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td>le(other[, level, fill_value, axis])</td>
<td>Return Less than or equal to of series and other, element-wise (binary operator ( \leq )).</td>
</tr>
<tr>
<td>lt(other[, level, fill_value, axis])</td>
<td>Return Less than of series and other, element-wise (binary operator ( \lt )).</td>
</tr>
<tr>
<td>mad([axis, skipna, level])</td>
<td>Return the mean absolute deviation of the values over the requested axis.</td>
</tr>
<tr>
<td>map(arg[, na_action])</td>
<td>Map values of Series according to an input mapping or function.</td>
</tr>
<tr>
<td>mask(cond[, other, inplace, axis, level, ...])</td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td>max([axis, skipna, level, numeric_only])</td>
<td>Return the maximum of the values over the requested axis.</td>
</tr>
<tr>
<td>mean([axis, skipna, level, numeric_only])</td>
<td>Return the mean of the values over the requested axis.</td>
</tr>
<tr>
<td>median([axis, skipna, level, numeric_only])</td>
<td>Return the median of the values over the requested axis.</td>
</tr>
<tr>
<td>memory_usage([index, deep])</td>
<td>Return the memory usage of the Series.</td>
</tr>
<tr>
<td>min([axis, skipna, level, numeric_only])</td>
<td>Return the minimum of the values over the requested axis.</td>
</tr>
<tr>
<td>mod(other[, level, fill_value, axis])</td>
<td>Return Modulo of series and other, element-wise (binary operator ( \mod )).</td>
</tr>
<tr>
<td>mode([dropna])</td>
<td>Return the mode(s) of the Series.</td>
</tr>
<tr>
<td>mul(other[, level, fill_value, axis])</td>
<td>Return Multiplication of series and other, element-wise (binary operator ( \times )).</td>
</tr>
<tr>
<td>nsmallest([n, keep])</td>
<td>Return the smallest ( n ) elements.</td>
</tr>
<tr>
<td>notna()</td>
<td>Detect existing (non-missing) values.</td>
</tr>
<tr>
<td>notnull()</td>
<td>Series.notnull is an alias for Series.notna.</td>
</tr>
<tr>
<td>nunique([dropna])</td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td>pad([axis, inplace, limit, downcast])</td>
<td>Synonym for DataFrame.fillna() with method='ffill'.</td>
</tr>
<tr>
<td>pct_change([periods, fill_method, limit, freq])</td>
<td>Percentage change between the current and a prior element.</td>
</tr>
<tr>
<td>pipe(func, *args, **kwargs)</td>
<td>Apply chainable functions that expect Series or DataFrames.</td>
</tr>
<tr>
<td>plot</td>
<td>alias of pandas.plotting._core.PlotAccessor</td>
</tr>
<tr>
<td>pop(item)</td>
<td>Return item and drops from series.</td>
</tr>
<tr>
<td>pow(other[, level, fill_value, axis])</td>
<td>Return Exponential power of series and other, element-wise (binary operator ( \times )).</td>
</tr>
<tr>
<td>prod([axis, skipna, level, numeric_only, ...])</td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td>product([axis, skipna, level, numeric_only, ...])</td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td>quantile([q, interpolation])</td>
<td>Return value at the given quantile.</td>
</tr>
</tbody>
</table>

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>radd(other[, level, fill_value, axis])</code></td>
<td>Return Addition of series and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>rank([axis, method, numeric_only, ...])</code></td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return the flattened underlying data as an ndarray.</td>
</tr>
<tr>
<td><code>rdiv(other[, level, fill_value, axis])</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>rdivmod(other[, level, fill_value, axis])</code></td>
<td>Return Integer division and modulo of series and other, element-wise (binary operator <code>rdivmod</code>).</td>
</tr>
<tr>
<td><code>reindex(*args, **kwargs)</code></td>
<td>Conform Series to new index with optional filling logic.</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit, ...])</code></td>
<td>Return an object with matching indices as other object.</td>
</tr>
<tr>
<td><code>rename([index, axis, copy, inplace, level, ...])</code></td>
<td>Alter Series index labels or name.</td>
</tr>
<tr>
<td><code>rename_axis([mapper, index, columns, axis, ...])</code></td>
<td>Set the name of the axis for the index or columns.</td>
</tr>
<tr>
<td><code>reorder_levels(order)</code></td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of a Series.</td>
</tr>
<tr>
<td><code>replace([to_replace, value, inplace, limit, ...])</code></td>
<td>Replace values given in <code>to_replace</code> with <code>value</code>.</td>
</tr>
<tr>
<td><code>resample(rule[, axis, closed, label, ...])</code></td>
<td>Resample time-series data.</td>
</tr>
<tr>
<td><code>reset_index([level, drop, name, inplace])</code></td>
<td>Generate a new DataFrame or Series with the index reset.</td>
</tr>
<tr>
<td><code>rfloordiv(other[, level, fill_value, axis])</code></td>
<td>Return Integer division of series and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>rmod(other[, level, fill_value, axis])</code></td>
<td>Return Modulo of series and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>rmul(other[, level, fill_value, axis])</code></td>
<td>Return Multiplication of series and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>rolling(window[, min_periods, center, ...])</code></td>
<td>Provide rolling window calculations.</td>
</tr>
<tr>
<td><code>round([decimals])</code></td>
<td>Round each value in a Series to the given number of decimals.</td>
</tr>
<tr>
<td><code>rpow(other[, level, fill_value, axis])</code></td>
<td>Return Exponential power of series and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>rsub(other[, level, fill_value, axis])</code></td>
<td>Return Subtraction of series and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>rtruediv(other[, level, fill_value, axis])</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>sample([n, frac, replace, weights, ...])</code></td>
<td>Return a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><code>searchsorted(value[, side, sorter])</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>sem([axis, skipna, level, ddof, numeric_only])</code></td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><code>set_axis(labels[, axis, inplace])</code></td>
<td>Assign desired index to given axis.</td>
</tr>
<tr>
<td><code>set_flags(*[, copy, allows_duplicate_labels])</code></td>
<td>Return a new object with updated flags.</td>
</tr>
<tr>
<td><code>shift([periods, freq, axis, fill_value])</code></td>
<td>Shift index by desired number of periods with an optional time <code>freq</code>.</td>
</tr>
<tr>
<td><code>skew([axis, skipna, level, numeric_only])</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>slice_shift([periods, axis])</code></td>
<td>(DEPRECATED) Equivalent to <code>shift</code> without copying data.</td>
</tr>
<tr>
<td><code>sort_index([axis, level, ascending, ...])</code></td>
<td>Sort Series by index labels.</td>
</tr>
<tr>
<td><code>sort_values([axis, ascending, inplace, ...])</code></td>
<td>Sort by the values.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sparse</code></td>
<td>Alias of <code>pandas.core.arrays.sparseAccessor.SparseAccessor</code></td>
</tr>
<tr>
<td><code>squeeze</code></td>
<td>Squeeze 1 dimensional axis objects into scalars.</td>
</tr>
<tr>
<td><code>std</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>str</code></td>
<td>Alias of <code>pandas.core.strings.accessor.StringMethods</code></td>
</tr>
<tr>
<td><code>sub</code></td>
<td>Return Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>subtract</code></td>
<td>Return Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>sum</code></td>
<td>Return the sum of the values over the requested axis.</td>
</tr>
<tr>
<td><code>swapaxes</code></td>
<td>Interchange axes and swap values axes appropriately.</td>
</tr>
<tr>
<td><code>swaplevel</code></td>
<td>Swap levels i and j in a <code>MultiIndex</code>.</td>
</tr>
<tr>
<td><code>tail</code></td>
<td>Return the last n rows.</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><code>to_clipboard</code></td>
<td>Copy object to the system clipboard.</td>
</tr>
<tr>
<td><code>to_csv</code></td>
<td>Write object to a comma-separated values (csv) file.</td>
</tr>
<tr>
<td><code>to_dict</code></td>
<td>Convert Series to {label -&gt; value} dict or dict-like object.</td>
</tr>
<tr>
<td><code>to_excel</code></td>
<td>Write object to an Excel sheet.</td>
</tr>
<tr>
<td><code>to_frame</code></td>
<td>Convert Series to DataFrame.</td>
</tr>
<tr>
<td><code>to_hdf</code></td>
<td>Write the contained data to an HDF5 file using HDF-Store.</td>
</tr>
<tr>
<td><code>to_json</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>to_latex</code></td>
<td>Render object to a LaTeX tabular, longtable, or nested table.</td>
</tr>
<tr>
<td><code>to_list</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>to_markdown</code></td>
<td>Print Series in Markdown-friendly format.</td>
</tr>
<tr>
<td><code>to_numpy</code></td>
<td>A NumPy ndarray representing the values in this Series or Index.</td>
</tr>
<tr>
<td><code>to_period</code></td>
<td>Convert Series from DatetimeIndex to PeriodIndex.</td>
</tr>
<tr>
<td><code>to_pickle</code></td>
<td>Pickle (serialize) object to file.</td>
</tr>
<tr>
<td><code>to_sql</code></td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td><code>to_string</code></td>
<td>Render a string representation of the Series.</td>
</tr>
<tr>
<td><code>to_timestamp</code></td>
<td>Cast to DatetimeIndex of Timestamps, at beginning of period.</td>
</tr>
<tr>
<td><code>to_xarray</code></td>
<td>Return an xarray object from the pandas object.</td>
</tr>
<tr>
<td><code>tolist</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>Call func on self producing a Series with the same axis shape as self.</td>
</tr>
<tr>
<td><code>transpose</code></td>
<td>Return the transpose, which is by definition self.</td>
</tr>
<tr>
<td><code>truediv</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><code>truncate</code></td>
<td>Truncate a Series or DataFrame before and after some index value.</td>
</tr>
<tr>
<td><code>tshift</code></td>
<td>(DEPRECATED) Shift the time index, using the index's frequency if available.</td>
</tr>
<tr>
<td><code>tz_convert</code></td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
</tbody>
</table>

**continues on next page**

---

**3.3. Series**

1199
pandas: powerful Python data analysis toolkit, Release 1.4.2

Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tz_localize</code></td>
<td>Localize tz-naive index of a Series or DataFrame to target time zone.</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Return unique values of Series object.</td>
</tr>
<tr>
<td><code>unstack()</code></td>
<td>Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.</td>
</tr>
<tr>
<td><code>update()</code></td>
<td>Modify Series in place using values from passed Series.</td>
</tr>
<tr>
<td><code>value_counts()</code></td>
<td>Return a Series containing counts of unique values.</td>
</tr>
<tr>
<td><code>var()</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>view()</code></td>
<td>Create a new view of the Series.</td>
</tr>
<tr>
<td><code>where()</code></td>
<td>Replace values where the condition is False.</td>
</tr>
<tr>
<td><code>xs()</code></td>
<td>Return cross-section from the Series/DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.Series.abs**

Series.abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs  Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute  Calculate the absolute value element-wise.

Notes

For complex inputs, $1.2 + 1j$, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```python
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1     2.00
2     3.33
3     4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```python
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0  1 days
dtype: timedelta64[ns]

Select rows with data closest to certain value using argsort (from StackOverflow).

```python
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
   a  b  c
0  4 10 100
1  5 20  50
2  6 30 -30
3  7 40 -50
```

```python
>>> df.loc[(df.c - 43).abs().argsort()]
   a  b  c
1  5 20  50
0  4 10 100
2  6 30 -30
3  7 40 -50
```

### pandas.Series.add

`Series.add(other, level=None, fill_value=None, axis=0)`

Return Addition of series and other, element-wise (binary operator `add`).

Equivalent to `series + other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **Series** The result of the operation.

**See also:**

- `Series.radd` Reverse of the Addition operator, see Python documentation for more details.
Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64

>>> a.add(b, fill_value=0)
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

**pandas.Series.add_prefix**

Series.add_prefix(prefix)

Prefix labels with string prefix.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters**

- prefix [str] The string to add before each label.

**Returns**

Series or DataFrame New Series or DataFrame with updated labels.

See also:

- **Series.add_suffix** Suffix row labels with string suffix.
- **DataFrame.add_suffix** Suffix column labels with string suffix.
## Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64

>>> s.add_prefix('item_')
item_0  1
item_1  2
item_2  3
item_3  4
dtype: int64

>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6

>>> df.add_prefix('col_')
col_A  col_B
0    1    3
1    2    4
2    3    5
3    4    6
```

### pandas.Series.add_suffix

```python
pandas.Series.add_suffix(suffix)
```

Suffix labels with string `suffix`.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters**

- `suffix` [str] The string to add after each label.

**Returns**

- **Series or DataFrame** New Series or DataFrame with updated labels.

**See also:**

- `Series.add_prefix` Prefix row labels with string `prefix`.
- `DataFrame.add_prefix` Prefix column labels with string `prefix`. 
Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64

>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64

>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6

>>> df.add_suffix('_col')
   A_col  B_col
0     1     3
1     2     4
2     3     5
3     4     6
```

pandas.Series.agg

Series.agg(func=None, axis=0, *args, **kwargs)

Aggregate using one or more operations over the specified axis.

Parameters

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. Accepted combinations are:
  - function
  - string function name
  - list of functions and/or function names, e.g. `[np.sum, 'mean']`
  - dict of axis labels -> functions, function names or list of such.
- **axis** [[0 or ‘index’]] Parameter needed for compatibility with DataFrame.
- **args** Positional arguments to pass to `func`.
**kwargs  Keyword arguments to pass to `func`.

Returns

**scalar, Series or DataFrame**  The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

- **Series.apply**  Invoke function on a Series.
- **Series.transform**  Transform function producing a Series with like indexes.

Notes

`agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See [Mutating with User Defined Function (UDF) methods](#) for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0  1
1  2
2  3
3  4
dtype: int64

>>> s.agg('min')
1

>>> s.agg(['min', 'max'])
min  1
max  4
dtype: int64
```
pandas.Series.aggregate

Series.aggregate(func=None, axis=0, *args, **kwargs)

Aggregate using one or more operations over the specified axis.

Parameters

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.
  
  Accepted combinations are:
  
  - function
  - string function name
  - list of functions and/or function names, e.g. [np.sum, \`mean\`]
  - dict of axis labels -> functions, function names or list of such.

- **axis** [[0 or `index`]] Parameter needed for compatibility with DataFrame.

- **args** Positional arguments to pass to func.

- **kwargs** Keyword arguments to pass to func.

Returns

- scalar, Series or DataFrame The return can be:
  
  - scalar : when Series.agg is called with single function
  - Series : when DataFrame.agg is called with a single function
  - DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

- **Series.apply** Invoke function on a Series.
- **Series.transform** Transform function producing a Series with like indexes.

Notes

agg is an alias for aggregate. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.
Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64

>>> s.agg('min')
1

>>> s.agg(['min', 'max'])
min  1
max  4
dtype: int64
```

`pandas.Series.align`

`Series.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0, broadcast_axis=None)`

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

**Parameters**

- `other` [DataFrame or Series]
- `join` [{'outer', 'inner', 'left', 'right'}, default 'outer']
- `axis` [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).
- `level` [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.
- `copy` [bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.
- `fill_value` [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any “compatible” value.
  - pad / ffill: propagate last valid observation forward to next valid.
  - backfill / bfill: use NEXT valid observation to fill gap.
- `limit` [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.
- `fill_axis` [[0 or ‘index’], default 0] Filling axis, method and limit.
**broadcast_axis** [[0 or ‘index’], default None] Broadcast values along this axis, if aligning two objects of different dimensions.

**Returns**

(left, right) [(Series, type of other)] Aligned objects.

**Examples**

```python
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1, 2])
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   A  B  C  D
0  1  2  3  4
1  6  7  8  9
>>> other
   A  B  C  D
2 10 20 30 40
3 60 70 80 90
4 600 700 800 900
Align on columns:
>>> left, right = df.align(other, join="outer", axis=1)
>>> left
   A  B  C  D  E
0 4 2  NaN 1 3
1 9 7  NaN 6 8
>>> right
   A  B  C  D  E
2 10 20 30 40  NaN
3 60 70 80 90  NaN
4 600 700 800 900  NaN
We can also align on the index:
>>> left, right = df.align(other, join="outer", axis=0)
>>> left
   D  B  E  A
0 1.0 2.0 3.0 4.0
1 6.0 7.0 8.0 9.0
2 NaN NaN NaN NaN
3 NaN NaN NaN NaN
4 NaN NaN NaN NaN
>>> right
   A  B  C  D
0 NaN NaN NaN NaN
1 NaN NaN NaN NaN
2 10.0 20.0 30.0 40.0
(continues on next page)"
Finally, the default `axis=None` will align on both index and columns:

```
left, right = df.align(other, join="outer", axis=None)

left
A  B  C  D  E
1  4.0 2.0 NaN 1.0 3.0
2  9.0 7.0 NaN 6.0 8.0
3  NaN NaN NaN NaN NaN
4  NaN NaN NaN NaN NaN

right
A  B  C  D  E
1  NaN NaN NaN NaN NaN
2 10.0 20.0 30.0 40.0 NaN
3 60.0 70.0 80.0 90.0 NaN
4 600.0 700.0 800.0 900.0 NaN
```

**pandas.Series.all**

`Series.all(axis=0, bool_only=None, skipna=True, level=None, **kwargs)`

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a DataFrame axis that is False or equivalent (e.g. zero or empty).

**Parameters**

- `axis` ([0 or ‘index’, 1 or ‘columns’, None], default 0] Indicate which axis or axes should be reduced.
  - 0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
  - 1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
  - None : reduce all axes, return a scalar.

- `bool_only` [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

- `skipna` [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

- `level` [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

- `**kwargs` [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- scalar or Series If level is specified, then, Series is returned; otherwise, scalar is returned.

**See also:**

3.3. Series
Series.all  Return True if all elements are True.

DataFrame.any  Return True if one (or more) elements are True.

Examples

Series

```python
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```python
>>> df = pd.DataFrame({
    'col1': [True, True],
    'col2': [True, False]
})

>>> df
   col1  col2
0  True  True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```python
>>> df.all()
   col1  col2
0   True  True
dtype: bool
```

Specify axis='columns' to check if row-wise values all return True.

```python
>>> df.all(axis='columns')
   0   True
   1  False
dtype: bool
```

Or axis=None for whether every value is True.

```python
>>> df.all(axis=None)
False
```
pandas.Series.any

Series.any(axis=0, bool_only=None, skipna=True, level=None, **kwargs)

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

- **axis** [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.
  - 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
  - 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
  - None : reduce all axes, return a scalar.

- **bool_only** [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

- **skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

- **kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

- scalar or Series If level is specified, then, Series is returned; otherwise, scalar is returned.

See also:

- numpy.any Numpy version of this method.
- Series.any Return whether any element is True.
- Series.all Return whether all elements are True.
- DataFrame.any Return whether any element is True over requested axis.
- DataFrame.all Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```python
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
DataFrame
Whether each column contains at least one True element (the default).
>>> df =
>>> df
A B
0 1 0
1 2 2

pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
C
0
0

>>> df.any()
A
True
B
True
C
False
dtype: bool
Aggregating over the columns.
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
A B
0
True 1
1 False 2
>>> df.any(axis='columns')
0
True
1
True
dtype: bool
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
A B
0
True 1
1 False 0
>>> df.any(axis='columns')
0
True
1
False
dtype: bool
Aggregating over the entire DataFrame with axis=None.
>>> df.any(axis=None)
True
any for an empty DataFrame is an empty Series.

1212

Chapter 3. API reference


pandas.Series.append

Series.append(to_append, ignore_index=False, verify_integrity=False)
Concatenate two or more Series.

Deprecated since version 1.4.0: Use concat() instead. For further details see Deprecated
DataFrame.append and Series.append

Parameters

- to_append [Series or list/tuple of Series] Series to append with self.
- ignore_index [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.
- verify_integrity [bool, default False] If True, raise Exception on creating index with duplicates.

Returns

Series Concatenated Series.

See also:

- concat General function to concatenate DataFrame or Series objects.

Notes

Iteratively appending to a Series can be more computationally intensive than a single concatenate. A better solution is to append values to a list and then concatenate the list with the original Series all at once.

Examples

```python
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3, 4, 5])
>>> s1.append(s2)
0 1
1 2
2 3
0 4
1 5
2 6
dtype: int64

>>> s1.append(s3)
0 1
1 2
2 3
3 4
```

(continues on next page)
With `ignore_index` set to True:

```python
>>> s1.append(s2, ignore_index=True)
    0  1
    1  2
    2  3
    3  4
    4  5
    5  6
dtype: int64
```

With `verify_integrity` set to True:

```python
>>> s1.append(s2, verify_integrity=True)
Traceback (most recent call last):
  ...  
ValueError: Indexes have overlapping values: [0, 1, 2]
```

**pandas.Series.apply**

```python
Series.apply(func, convert_dtype=True, args=(), **kwargs)
```

Invoke function on values of Series.

Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values.

**Parameters**

- `func` [function] Python function or NumPy ufunc to apply.
- `convert_dtype` [bool, default True] Try to find better dtype for elementwise function results. If False, leave as dtype=object. Note that the dtype is always preserved for some extension array dtypes, such as Categorical.
- `args` [tuple] Positional arguments passed to func after the series value.
- `**kwargs` Additional keyword arguments passed to func.

**Returns**

- `Series` or `DataFrame` If func returns a Series object the result will be a DataFrame.

**See also:**

- `Series.map` For element-wise operations.
- `Series.agg` Only perform aggregating type operations.
- `Series.transform` Only perform transforming type operations.
Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

Examples

Create a series with typical summer temperatures for each city.

```python
>>> s = pd.Series([20, 21, 12],
                 index=['London', 'New York', 'Helsinki'])
>>> s
London    20
New York  21
Helsinki  12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```python
>>> def square(x):
...     return x ** 2
>>> s.apply(square)
London    400
New York  441
Helsinki  144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```python
>>> s.apply(lambda x: x ** 2)
London    400
New York  441
Helsinki  144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```python
>>> def subtract_custom_value(x, custom_value):
...     return x - custom_value
>>> s.apply(subtract_custom_value, args=(5,))
London    15
New York  16
Helsinki  7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply()`.

```python
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x += kwargs[month]
...     return x
```

3.3. Series
```python
>>> s.apply(add_custom_values, june=30, july=20, august=25)
London    95
New York  96
Helsinki  87
dtype: int64
```

Use a function from the Numpy library.

```python
>>> s.apply(np.log)
London    2.995732
New York  3.044522
Helsinki  2.484907
dtype: float64
```

**pandas.Series.argmax**

```python
Series.argmax(axis=None, skipna=True, *args, **kwargs)
```

Return int position of the largest value in the Series.

If the maximum is achieved in multiple locations, the first row position is returned.

**Parameters**

- `axis` [{None}] Dummy argument for consistency with Series.
- `skipna` [bool, default True] Exclude NA/null values when showing the result.
- `*args, **kwargs` Additional arguments and keywords for compatibility with NumPy.

**Returns**

- `int` Row position of the maximum value.

**See also:**

- `Series.argmax` Return position of the maximum value.
- `Series.argmin` Return position of the minimum value.
- `numpy.ndarray.argmax` Equivalent method for numpy arrays.
- `Series.idxmax` Return index label of the maximum values.
- `Series.idxmin` Return index label of the minimum values.

**Examples**

Consider dataset containing cereal calories

```python
>>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
...                 'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes    100.0
Almond Delight 110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff     110.0
dtype: float64
```
The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

**pandas.Series.argmin**

Series.argmin(axis=None, skipna=True, *args, **kwargs)

Return int position of the smallest value in the Series. If the minimum is achieved in multiple locations, the first row position is returned.

**Parameters**

- **axis** ([None]) Dummy argument for consistency with Series.
- **skipna** [bool, default True] Exclude NA/null values when showing the result.
- **args**, **kwargs** Additional arguments and keywords for compatibility with NumPy.

**Returns**

- **int** Row position of the minimum value.

**See also:**

- **Series.argmin** Return position of the minimum value.
- **Series.argmax** Return position of the maximum value.
- **numpy.ndarray.argmin** Equivalent method for numpy arrays.
- **Series.idxmax** Return index label of the maximum values.
- **Series.idxmin** Return index label of the minimum values.

**Examples**

Consider dataset containing cereal calories

```python
>>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
... 'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes    100.0
Almond Delight 110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff 110.0
dtype: float64
```

```python
>>> s.argmax()
2
>>> s.argmin()
0
```
The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

**pandas.Series.argsort**

Series.argsort\( (axis=0, \text{kind}='\text{quicksort}', \text{order}=\text{None}) \)

Return the integer indices that would sort the Series values.

Override ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values.

**Parameters**

\[ \text{axis} \ [0 \text{ or } \text{“index”}]] \text{ Has no effect but is accepted for compatibility with numpy.} \]

\[ \text{kind} \ [\text{“mergesort”, “quicksort”, “heapsort”, “stable”}, \text{ default “quicksort”}] \text{ Choice of sorting algorithm. See } \text{numpy.sort()} \text{ for more information. “mergesort” and “stable” are the only stable algorithms.} \]

\[ \text{order} \ [\text{None}] \text{ Has no effect but is accepted for compatibility with numpy.} \]

**Returns**

Series[np.intp] Positions of values within the sort order with -1 indicating nan values.

See also:

\text{numpy.ndarray.argsort} Returns the indices that would sort this array.

**pandas.Series.asfreq**

Series.asfreq\( (freq, \text{method}=\text{None}, \text{how}=\text{None}, \text{normalize}=\text{False}, \text{fill_value}=\text{None}) \)

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this Series is a PeriodIndex, the new index is the result of transforming the original index with PeriodIndex.asfreq (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to pd.date_range(start, end, freq=freq) where start and end are, respectively, the first and last entries in the original index (see pandas.date_range()). The values corresponding to any timesteps in the new index which were not present in the original index will be null (NaN), unless a method for filling such unknowns is provided (see the method parameter below).

The resample() method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

**Parameters**

\[ \text{freq} \ [\text{DateOffset or str]} \text{ Frequency DateOffset or string.} \]

\[ \text{method} \ [\text{“backfill”}/’bfill’, “pad”]/’ffill’, \text{ default None}] \text{ Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):} \]

\[ \cdot \text{ “pad”}/’ffill’: \text{ propagate last valid observation forward to next valid} \]

\[ \cdot \text{ “backfill”}/’bfill’: \text{ use NEXT valid observation to fill.} \]

\[ \text{how} \ [\text{“start”, “end”}, \text{ default end}] \text{ For PeriodIndex only (see PeriodIndex.asfreq).} \]
normalize [bool, default False] Whether to reset output index to midnight.

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

Series Series object reindexed to the specified frequency.

See also:

reindex Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
          s
2000-01-01 00:00:00  0.0
2000-01-01 00:01:00  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:03:00  3.0
```

Upsample the series into 30 second bins.

```python
>>> df.asfreq(freq='30S')
          s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  NaN
2000-01-01 00:03:00  3.0
```

Upsample again, providing a fill value.

```python
>>> df.asfreq(freq='30S', fill_value=9.0)
          s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  9.0
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  9.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  9.0
2000-01-01 00:03:00  3.0
```
Upsample again, providing a method.

```python
>>> df.asfreq(freq='30S', method='bfill')
   s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```

**pandas.Series.asof**

`Series.asof(where, subset=None)`

Return the last row(s) without any NaNs before `where`.

The last row (for each element in `where`, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not `None`).

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame.

**Parameters**

- `where` [date or array-like of dates] Date(s) before which the last row(s) are returned.
- `subset` [str or array-like of str, default `None`] For DataFrame, if not `None`, only use these columns to check for NaNs.

**Returns**

- scalar, Series, or DataFrame The return can be:
  - scalar : when `self` is a Series and `where` is a scalar
  - Series: when `self` is a Series and `where` is an array-like, or when `self` is a DataFrame and `where` is a scalar
  - DataFrame : when `self` is a DataFrame and `where` is an array-like

**See also:**

- `merge_asof` Perform an asof merge. Similar to left join.

**Notes**

Dates are assumed to be sorted. Raises if this is not the case.
Examples

A Series and a scalar `where`.

```python
def asof(20)
s.asof(20)
```

For a sequence `where`, a Series is returned. The first value is NaN, because the first element of `where` is before the first index value.

```python
def asof([5, 20])
s.asof([5, 20])
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```python
def asof(30)
s.asof(30)
```

Take all columns into consideration.

```python
def asof(pd.DatetimeIndex(['2018-02-27 09:01:00', '2018-02-27 09:02:00', '2018-02-27 09:03:00', '2018-02-27 09:04:00', '2018-02-27 09:05:00']))
df.asof(pd.DatetimeIndex(['2018-02-27 09:01:00', '2018-02-27 09:02:00', '2018-02-27 09:03:00', '2018-02-27 09:04:00', '2018-02-27 09:05:00']))
```

Take a single column into consideration.

```python
def asof(pd.DatetimeIndex(['2018-02-27 09:01:00', '2018-02-27 09:02:00', '2018-02-27 09:03:00', '2018-02-27 09:04:00', '2018-02-27 09:05:00']))
df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30', '2018-02-27 09:04:30']))
```

```python
def asof(pd.DatetimeIndex(['2018-02-27 09:01:00', '2018-02-27 09:02:00', '2018-02-27 09:03:00', '2018-02-27 09:04:00', '2018-02-27 09:05:00']))
```
pandas.Series.astype

Series.astype(dtype, copy=True, errors='raise')

Cast a pandas object to a specified dtype dtype.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame’s columns to column-specific types.

copy [bool, default True] Return a copy when copy=True (be very careful setting copy=False as changes to values then may propagate to other pandas objects).

errors [{‘raise’, ‘ignore’}, default ‘raise’] Control raising of exceptions on invalid data for provided dtype.

• raise: allow exceptions to be raised
• ignore: suppress exceptions. On error return original object.

Returns

casted [same type as caller]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

numpy.ndarray.astype Cast a numpy array to a specified type.

Notes

Deprecated since version 1.3.0: Using astype to convert from timezone-naive dtype to timezone-aware dtype is deprecated and will raise in a future version. Use Series.dt.tz_localize() instead.

Examples

Create a DataFrame:

```python
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
coll    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```python
>>> df.astype('int32').dtypes
coll    int32
col2    int32
dtype: object
```
Cast col1 to int32 using a dictionary:

```python
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```python
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```python
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```python
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(...
...    categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using copy=False and changing data on a new pandas object may propagate changes:

```python
>>> s1 = pd.Series([1, 2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1    2
dtype: int64
```

Create a series of dates:

```python
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0  2020-01-01
1  2020-01-02
```

(continues on next page)
pandas.Series.at_time

Series.at_time(time, asof=False, axis=None)
Select values at particular time of day (e.g., 9:30AM).

Parameters
- time [datetime.time or str]
- axis [{0 or ‘index’, 1 or ‘columns’}, default 0]

Returns
Series or DataFrame

Raises
TypeError If the index is not a DatetimeIndex

See also:
between_time  Select values between particular times of the day.
first  Select initial periods of time series based on a date offset.
last  Select final periods of time series based on a date offset.
DatetimeIndex.indexer_at_time  Get just the index locations for values at particular time of the day.

Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
```
```
2018-04-09 00:00:00 1
2018-04-09 12:00:00 2
2018-04-10 00:00:00 3
2018-04-10 12:00:00 4
```

```python
>>> ts.at_time('12:00')
```
```
2018-04-09 12:00:00 2
2018-04-10 12:00:00 4
```
pandas.Series.autocorr

Series.autocorr(lag=1)

Compute the lag-N autocorrelation.
This method computes the Pearson correlation between the Series and its shifted self.

Parameters

    lag [int, default 1] Number of lags to apply before performing autocorrelation.

Returns

    float The Pearson correlation between self and self.shift(lag).

See also:

Series.corr Compute the correlation between two Series.
Series.shift Shift index by desired number of periods.
DataFrame.corr Compute pairwise correlation of columns.
DataFrame.corrwith Compute pairwise correlation between rows or columns of two DataFrame objects.

Notes

If the Pearson correlation is not well defined return ‘NaN’.

Examples

```python
>>> s = pd.Series([0.25, 0.5, 0.2, -0.05])
>>> s.autocorr()
0.10355...
>>> s.autocorr(lag=2)
-0.99999...
```

If the Pearson correlation is not well defined, then ‘NaN’ is returned.

```python
>>> s = pd.Series([1, 0, 0, 0])
>>> s.autocorr()
nan
```

pandas.Series.backfill

Series.backfill(axis=None, inplace=False, limit=None, downcast=None)

Synonym for DataFrame.fillna() with method='bfill'.

Returns

Series/DataFrame or None  Object with missing values filled or None if inplace=True.
**pandas.Series.between**

Series.between(left, right, inclusive='both')

Return boolean Series equivalent to left <= series <= right.

This function returns a boolean vector containing `True` wherever the corresponding Series element is between the boundary values `left` and `right`. NA values are treated as `False`.

**Parameters**

- **left** [scalar or list-like] Left boundary.
- **right** [scalar or list-like] Right boundary.
- **inclusive** ["both", "neither", "left", "right"] Include boundaries. Whether to set each bound as closed or open.

Changed in version 1.3.0.

**Returns**

Series Series representing whether each element is between left and right (inclusive).

**See also:**

- **Series.gt** Greater than of series and other.
- **Series.lt** Less than of series and other.

**Notes**

This function is equivalent to `(left <= ser) & (ser <= right)`

**Examples**

```python
>>> s = pd.Series([2, 0, 4, 8, np.nan])
>>> s.between(1, 4)
0    True
1   False
2    True
3   False
4   False
dtype: bool
```

Boundary values are included by default:

```python
>>> s.between(1, 4, inclusive="neither")
0    True
1   False
2    True
3   False
4   False
dtype: bool
```

With `inclusive` set to "neither" boundary values are excluded:
left and right can be any scalar value:

```python
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0   False
1    True
2    True
3   False
dtype: bool
```

**pandas.Series.between_time**

Series.between_time(start_time, end_time, include_start=NoDefault.no_default, include_end=NoDefault.no_default, inclusive=None, axis=None)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting start_time to be later than end_time, you can get the times that are not between the two times.

**Parameters**

- **start_time** [datetime.time or str] Initial time as a time filter limit.
- **end_time** [datetime.time or str] End time as a time filter limit.
- **include_start** [bool, default True] Whether the start time needs to be included in the result.
  
  Deprecated since version 1.4.0: Arguments include_start and include_end have been deprecated to standardize boundary inputs. Use inclusive instead, to set each bound as closed or open.

- **include_end** [bool, default True] Whether the end time needs to be included in the result.
  
  Deprecated since version 1.4.0: Arguments include_start and include_end have been deprecated to standardize boundary inputs. Use inclusive instead, to set each bound as closed or open.

- **inclusive** ["both", “neither”, “left”, “right"], default “both”] Include boundaries; whether to set each bound as closed or open.

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value.

**Returns**

Series or DataFrame Data from the original object filtered to the specified dates range.

**Raises**

TypeError If the index is not a DatetimeIndex.

**See also:**

- at_time Select values at a particular time of the day.
- first Select initial periods of time series based on a date offset.
- last Select final periods of time series based on a date offset.
**DatetimeIndex.indexer_between_time** Get just the index locations for values between particular times of the day.

**Examples**

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
   A
2018-04-09  00:00:00  1
2018-04-10  00:20:00  2
2018-04-11  00:40:00  3
2018-04-12  01:00:00  4

>>> ts.between_time('0:15', '0:45')
   A
2018-04-10  00:20:00  2
2018-04-11  00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```python
>>> ts.between_time('0:45', '0:15')
   A
2018-04-09  00:00:00  1
2018-04-12  01:00:00  4
```

**pandas.Series.bfill**

Series.bfill(*axis=None, inplace=False, limit=None, downcast=None*)

Synonym for `DataFrame.fillna()` with `method='bfill'`.

**Returns**

Series/DataFrame or None Object with missing values filled or None if `inplace=True`.

**pandas.Series.bool**

Series.bool()

Return the bool of a single element Series or DataFrame.

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

**Returns**

bool The value in the Series or DataFrame.

**See also:**

Series.astype Change the data type of a Series, including to boolean.

DataFrame.astype Change the data type of a DataFrame, including to boolean.
**numpy.bool_** NumPy boolean data type, used by pandas for boolean values.

**Examples**

The method will only work for single element objects with a boolean value:

```python
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```
dtype: category
Categories (3, object): ['c', 'b', 'a']

```python
>>> s.cat.reorder_categories(list("cba"))
  0  a
  1  b
  2  b
  3  c
  4  c
  5  c
dtype: category
Categories (3, object): ['c', 'b', 'a']
```

```python
>>> s.cat.add_categories(["d", "e")
  0  a
  1  b
  2  b
  3  c
  4  c
  5  c
dtype: category
Categories (5, object): ['a', 'b', 'c', 'd', 'e']
```

```python
>>> s.cat.remove_categories(["a", "c")
  0   NaN
  1   b
  2   b
  3  NaN
  4  NaN
  5  NaN
dtype: category
Categories (1, object): ['b']
```

```python
>>> s1 = s.cat.add_categories(["d", "e")
>>> s1.cat.remove_unused_categories()
  0  a
  1  b
  2  b
  3  c
  4  c
  5  c
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

```python
>>> s.cat.set_categories(list("abcde"))
  0  a
  1  b
  2  b
  3  c
  4  c
  5  c
```

(continues on next page)
dtype: category
Categories (5, object): ['a', 'b', 'c', 'd', 'e']

```python
>>> s.cat.as_ordered()
0   a
1   b
2   b
3   c
4   c
5   c
dtype: category
Categories (3, object): ['a' < 'b' < 'c']
```

```python
>>> s.cat.as_unordered()
0   a
1   b
2   b
3   c
4   c
5   c
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

**pandas.Series.clip**

`Series.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)`

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

**Parameters**

- `lower` [float or array-like, default None] Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.
- `upper` [float or array-like, default None] Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.
- `axis` [int or str axis name, optional] Align object with lower and upper along the given axis.
- `inplace` [bool, default False] Whether to perform the operation in place on the data.
- `*args, **kwargs` Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns**

- `Series` or `DataFrame` or `None` Same type as calling object with the values outside the clip boundaries replaced or None if `inplace=True`.

**See also:**

- `Series.clip` Trim values at input threshold in series.
DataFrame.clip  Trim values at input threshold in dataframe.
numpy.clip  Clip (limit) the values in an array.

Examples

```python
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0     9    -2
1    -3    -7
2     0     6
3    -1     8
4     5    -5
```

Clips per column using lower and upper thresholds:

```python
>>> df.clip(-4, 6)
   col_0  col_1
0     6    -2
1    -3    -4
2     0     6
3    -1     6
4     5    -4
```

Clips using specific lower and upper thresholds per column element:

```python
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0   2
1  -4
2  -1
3   6
4   3
dtype: int64
```

```python
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0     6     2
1    -3    -4
2     0     3
3     6     8
4     5     3
```

Clips using specific lower threshold per column element, with missing values:

```python
>>> t = pd.Series([2, -4, np.nan, 6, 3])
```

(continues on next page)
```python
4  3.0
dtype: float64
```

```python
>>> df.clip(t, axis=0)
col_0  col_1
0  9  2
1 -3 -4
2  0  6
3  6  8
4  5  3
```

### pandas.Series.combine

**Series.combine(other, func, fill_value=None)**

Combine the Series with a Series or scalar according to `func`.

Combine the Series and `other` using `func` to perform elementwise selection for combined Series. `fill_value` is assumed when value is missing at some index from one of the two objects being combined.

**Parameters**

- **other** [Series or scalar] The value(s) to be combined with the Series.
- **func** [function] Function that takes two scalars as inputs and returns an element.
- **fill_value** [scalar, optional] The value to assume when an index is missing from one Series or the other. The default specifies to use the appropriate NaN value for the underlying dtype of the Series.

**Returns**

- **Series** The result of combining the Series with the other object.

**See also:**

- **Series.combine_first** Combine Series values, choosing the calling Series’ values first.

**Examples**

Consider 2 Datasets `s1` and `s2` containing highest clocked speeds of different birds.

```python
>>> s1 = pd.Series({'falcon': 330.0, 'eagle': 160.0})
>>> s1
falcon 330.0
eagle 160.0
dtype: float64
>>> s2 = pd.Series({'falcon': 345.0, 'eagle': 200.0, 'duck': 30.0})
>>> s2
falcon 345.0
eagle 200.0
duck 30.0
dtype: float64
```

Now, to combine the two datasets and view the highest speeds of the birds across the two datasets
In the previous example, the resulting value for duck is missing, because the maximum of a NaN and a float is a NaN. So, in the example, we set `fill_value=0`, so the maximum value returned will be the value from some dataset.

```python
>>> s1.combine(s2, max, fill_value=0)
duck    30.0
eagle   200.0
falcon  345.0
dtype: float64
```

**pandas.Series.combine_first**

Series\texttt{.combine\_first}(other)

Update null elements with value in the same location in ‘other’.

Combine two Series objects by filling null values in one Series with non-null values from the other Series. Result index will be the union of the two indexes.

**Parameters**

- \texttt{other} [Series] The value(s) to be used for filling null values.

**Returns**

- Series The result of combining the provided Series with the other object.

**See also:**

- \texttt{Series.combine} Perform element-wise operation on two Series using a given function.

**Examples**

```python
>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4, 5])
>>> s1.combine_first(s2)
0  1.0
1  4.0
2  5.0
dtype: float64
```

Null values still persist if the location of that null value does not exist in \texttt{other}

```python
>>> s1 = pd.Series({'falcon': np.nan, 'eagle': 160.0})
>>> s2 = pd.Series({'eagle': 200.0, 'duck': 30.0})
>>> s1.combine_first(s2)
duck  30.0
eagle 160.0
```
pandas.Series.compare

Series.compare(other, align_axis=1, keep_shape=False, keep_equal=False)

Compare to another Series and show the differences.

New in version 1.1.0.

Parameters

other [Series] Object to compare with.

align_axis [0 or ‘index’, 1 or ‘columns’], default 1] Determine which axis to align the comparison on.

• 0, or ‘index’ [Resulting differences are stacked vertically] with rows drawn alternately from self and other.

• 1, or ‘columns’ [Resulting differences are aligned horizontally] with columns drawn alternately from self and other.

keep_shape [bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

keep_equal [bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

Returns

Series or DataFrame If axis is 0 or ‘index’ the result will be a Series. The resulting index will be a MultiIndex with ‘self’ and ‘other’ stacked alternately at the inner level.

If axis is 1 or ‘columns’ the result will be a DataFrame. It will have two columns namely ‘self’ and ‘other’.

See also:

DataFrame.compare Compare with another DataFrame and show differences.

Notes

Matching NaNs will not appear as a difference.

Examples

```python
>>> s1 = pd.Series(['a', 'b', 'c', 'd', 'e'])
>>> s2 = pd.Series(['a', 'a', 'c', 'b', 'e'])
```

Align the differences on columns
```python
>>> s1.compare(s2)
   self  other
1     b     a
3     d     b

Stack the differences on indices
```}

```python
>>> s1.compare(s2, align_axis=0)
   self     other
1     b     a
3     d     b
dtype: object

Keep all original rows
```}

```python
>>> s1.compare(s2, keep_shape=True)
   self     other
0  NaN  NaN
1     b     a
2  NaN  NaN
3     d     b
4  NaN  NaN

Keep all original rows and also all original values
```}

```python
>>> s1.compare(s2, keep_shape=True, keep_equal=True)
   self     other
0     a     a
1     b     a
2     c     c
3     d     b
4     e     e
```

**pandas.Series.convert_dtypes**

`Series.convert_dtypes(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True)`

Convert columns to best possible dtypes using dtypes supporting `pd.NA`.

New in version 1.0.0.

**Parameters**

- `infer_objects` [bool, default True] Whether object dtypes should be converted to the best possible types.
- `convert_string` [bool, default True] Whether object dtypes should be converted to `StringDtype()`.
- `convert_integer` [bool, default True] Whether, if possible, conversion can be done to integer extension types.
- `convert_boolean` [bool, defaults True] Whether object dtypes should be converted to `BooleanDtypes()`.
convert_floating [bool, defaults True] Whether, if possible, conversion can be done to floating extension types. If convert_integer is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

New in version 1.2.0.

Returns
Series or DataFrame Copy of input object with new dtype.

See also:
infer_objects Infer dtypes of objects.
to_datetime Convert argument to datetime.
to_timedelta Convert argument to timedelta.
to_numeric Convert argument to a numeric type.

Notes
By default, convert_dtypes will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support pd.NA. By using the options convert_string, convert_integer, convert_boolean and convert_boolean, it is possible to turn off individual conversions to StringDtype, the integer extension types, BooleanDtype or floating extension types, respectively.

For object-dtyped columns, if infer_objects is True, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to StringDtype, BooleanDtype or an appropriate integer or floating extension type, otherwise leave as object.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support pd.NA, the results of this method will change to support those new dtypes.

Examples

```python
>>> df = pd.DataFrame(
...     {  
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),  
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),  
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),  
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),  
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),  
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )
```

Start with a DataFrame with default dtypes.
Convert the DataFrame to use best possible dtypes.

```python
>>> dfd = df.convert_dtypes()
>>> dfd
 a   b    c    d   e    f
0   1.0  x   True  h  10.0  NaN
1   2.0  y  False  i   NaN  100.5
2   NaN  z     NaN  NaN  20.0  200.0
```

Start with a Series of strings and missing data represented by `np.nan`.

```python
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0   a
1   b
2  NaN
```

Obtain a Series with dtype `StringDtype`.

```python
>>> s.convert_dtypes()
0   a
1   b
2  <NA>
```

pandas.Series.copy

Series.copy(deep=True)

Make a copy of this object’s indices and data.

When deep=True (default), a new object will be created with a copy of the calling object’s data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When deep=False, a new object will be created without copying the calling object’s data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

- deep [bool, default True] Make a deep copy, including a copy of the data and the indices.
  With deep=False neither the indices nor the data are copied.

Returns

- copy [Series or DataFrame] Object type matches caller.

Notes

When deep=True, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to copy.deepcopy in the Standard Library, which recursively copies object data (see examples below).

While Index objects are copied when deep=True, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```python
>>> s = pd.Series([1, 2], index=['a', 'b'])
>>> s
a  1  
b  2  
dtype: int64

>>> s_copy = s.copy()
>>> s_copy
a  1  
b  2  
dtype: int64

Shallow copy versus default (deep) copy:

```
Deep copy has own copy of data and index.

```python
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```python
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a 3
b 4
dtype: int64
>>> shallow
a 3
b 4
dtype: int64
>>> deep
a 1
b 2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```python
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0 [10, 2]
1 [3, 4]
dtype: object
>>> deep
0  [10, 2]
1  [3, 4]
dtype: object
```
pandas.Series.corr

Series.corr(other, method='pearson', min_periods=None)
Compute correlation with other Series, excluding missing values.

Parameters

other [Series] Series with which to compute the correlation.
method [[{'pearson', 'kendall', 'spearman'} or callable] Method used to compute correlation:
  • pearson : Standard correlation coefficient
  • kendall : Kendall Tau correlation coefficient
  • spearman : Spearman rank correlation
  • callable: Callable with input two 1d ndarrays and returning a float.

min_periods [int, optional] Minimum number of observations needed to have a valid result.

Returns

float Correlation with other.

See also:

DataFrame.corr Compute pairwise correlation between columns.
DataFrame.corrwith Compute pairwise correlation with another DataFrame or Series.

Examples

```python
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> s1 = pd.Series([.2, .0, .6, .2])
>>> s2 = pd.Series([.3, .6, .0, .1])
>>> s1.corr(s2, method=histogram_intersection)
0.3
```
**pandas.Series.count**

Series.count(level=None)

Return number of non-NA/null observations in the Series.

Parameters

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series.

Returns

int or Series (if level specified) Number of non-null values in the Series.

See also:

*DataFrame.count* Count non-NA cells for each column or row.

Examples

```python
>>> s = pd.Series([0.0, 1.0, np.nan])
>>> s.count()
2
```

**pandas.Series.cov**

Series.cov(other, min_periods=None, ddof=1)

Compute covariance with Series, excluding missing values.

Parameters

other [Series] Series with which to compute the covariance.

min_periods [int, optional] Minimum number of observations needed to have a valid result.

ddof [int, default 1] Delta degrees of freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.

New in version 1.1.0.

Returns

float Covariance between Series and other normalized by N-1 (unbiased estimator).

See also:

*DataFrame.cov* Compute pairwise covariance of columns.
Examples

```python
>>> s1 = pd.Series([0.90010907, 0.13484424, 0.62036035])
>>> s2 = pd.Series([0.12528585, 0.26962463, 0.51111198])
>>> s1.cov(s2)
-0.01685762652715874
```

pandas.Series.cummax

```
Series.cummax(axis=None, skipna=True, *args, **kwargs)
```

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **scalar or Series** Return cumulative maximum of scalar or Series.

See also:

- **core.window.Expanding.max** Similar functionality but ignores NaN values.
- **Series.max** Return the maximum over Series axis.
- **Series.cummax** Return cumulative maximum over Series axis.
- **Series.cummin** Return cumulative minimum over Series axis.
- **Series.cumsum** Return cumulative sum over Series axis.
- **Series.cumprod** Return cumulative product over Series axis.

Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.
```python
>>> s.cummax()
0   2.0
1   NaN
2   5.0
3   5.0
4   5.0
dtype: float64

To include NA values in the operation, use `skipna=False`

```python
>>> s.cummax(skipna=False)
0   2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```

**DataFrame**

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0],
...                     columns=list('AB'))

>>> df
df

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cummax()

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>3.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

To iterate over columns and find the maximum in each row, use `axis=1`

```python
>>> df.cummax(axis=1)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
```
pandas.Series.cummin

Series.cummin(axis=None, skipna=True, *args, **kwargs)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis ([0 or ‘index’, 1 or ‘columns’], default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.

skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

scalar or Series Return cumulative minimum of scalar or Series.

See also:

core.window.Expanding.min Similar functionality but ignores NaN values.
Series.min Return the minimum over Series axis.
Series.cummax Return cumulative maximum over Series axis.
Series.cummin Return cumulative minimum over Series axis.
Series.cumsum Return cumulative sum over Series axis.
Series.cumprod Return cumulative product over Series axis.

Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
```

```diff
0   2.0  
1   NaN  
2   5.0  
3  -1.0  
4   0.0  
```
dtype: float64

By default, NA values are ignored.

```python
>>> s.cummin()
0   2.0  
1   NaN  
2   2.0  
3  -1.0  
4  -1.0  
```
dtype: float64

To include NA values in the operation, use skipna=False
s.cummin(skipna=False)

```
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

**DataFrame**

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                     columns=list('AB'))
>>> df
   A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
   A  B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
   A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

**pandas.Series.cumprod**

```
Series.cumprod(axis=None, skipna=True, *args, **kwargs)
```

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

**Parameters**

- `axis` ([0 or ‘index’, 1 or ‘columns’], default 0) The index or the name of the axis. 0 is equivalent to None or ‘index’.
- `skipna` [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- `*args, **kwargs` Additional keywords have no effect but might be accepted for compatibility with NumPy.
Returns

scalar or Series  Return cumulative product of scalar or Series.

See also:

core.window.Expanding.prod  Similar functionality but ignores NaN values.

Series.prod  Return the product over Series axis.

Series.cummax  Return cumulative maximum over Series axis.

Series.cummin  Return cumulative minimum over Series axis.

Series.cumsum  Return cumulative sum over Series axis.

Series.cumprod  Return cumulative product over Series axis.

Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
```
```bash
0 2.0
1 NaN
2 5.0
3 -1.0
4 0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cumprod()
```
```bash
0 2.0
1 NaN
2 10.0
3 -10.0
4 -0.0
dtype: float64
```

To include NA values in the operation, use skipna=False

```python
>>> s.cumprod(skipna=False)
```
```bash
0 2.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64
```

DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                     columns=list('AB'))
```
(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> df
   A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

>>> df.cumprod()
   A  B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0

To iterate over columns and find the product in each row, use `axis=1`

>>> df.cumprod(axis=1)
   A  B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0

**pandas.Series.cumsum**

Series.cumsum(axis=None, skipna=True, *args, **kwargs)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters**

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **scalar or Series** Return cumulative sum of scalar or Series.

See also:

- **core.window.Expanding.sum** Similar functionality but ignores NaN values.
- **Series.sum** Return the sum over Series axis.
- **Series.cummax** Return cumulative maximum over Series axis.
- **Series.cummin** Return cumulative minimum over Series axis.
- **Series.cumsum** Return cumulative sum over Series axis.
- **Series.cumprod** Return cumulative product over Series axis.
Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
```
```python
0   2.0
1   NaN
2   5.0
3  -1.0
4    0
dtype: float64
```
By default, NA values are ignored.

```python
>>> s.cumsum()
```
```python
0   2.0
1   NaN
2   7.0
3   6.0
4   6.0
dtype: float64
```
To include NA values in the operation, use `skipna=False`

```python
>>> s.cumsum(skipna=False)
```
```python
0   2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```

DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                    columns=list('AB'))
```
```python
A   B
0  2.0  1.0
1  3.0   NaN
2  1.0  0.0
```
By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cumsum()
```
```python
A   B
0  2.0  1.0
1  5.0   NaN
2  6.0  1.0
```
To iterate over columns and find the sum in each row, use `axis=1`
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.cumsum(axis=1)
   A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

**pandas.Series.describe**

Series.describe(*percentiles=None, include=None, exclude=None, datetime_is_numeric=False*)

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters**

- **percentiles** [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is [.25, .5, .75], which returns the 25th, 50th, and 75th percentiles.

- **include** ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for Series. Here are the options:
  - ‘all’ : All columns of the input will be included in the output.
  - A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit numpy.number. To limit it instead to object columns submit the numpy.object data type. Strings can also be used in the style of select_dtypes (e.g. df.describe(include=['O'])). To select pandas categorical columns, use 'category'
  - None (default) : The result will include all numeric columns.

- **exclude** [list-like of dtypes or None (default), optional] A black list of data types to omit from the result. Ignored for Series. Here are the options:
  - A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit numpy.number. To exclude object columns submit the data type numpy.object. Strings can also be used in the style of select_dtypes (e.g. df.describe(exclude=['O'])). To exclude pandas categorical columns, use 'category'
  - None (default) : The result will exclude nothing.

- **datetime_is_numeric** [bool, default False] Whether to treat datetime dtypes as numeric. This affects statistics calculated for the column. For DataFrame input, this also controls whether datetime columns are included by default.

**Returns**

- Series or DataFrame Summary statistics of the Series or Dataframe provided.

**See also:**

- DataFrame.count Count number of non-NA/null observations.
**DataFrame.max** Maximum of the values in the object.

**DataFrame.min** Minimum of the values in the object.

**DataFrame.mean** Mean of the values.

**DataFrame.std** Standard deviation of the observations.

**DataFrame.select_dtypes** Subset of a DataFrame including/excluding columns based on their dtype.

### Notes

For numeric data, the result’s index will include count, mean, std, min, max as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result’s index will include count, unique, top, and freq. The top is the most common value. The freq is the most common value’s frequency. Timestamps also include the first and last items.

If multiple object values have the highest count, then the count and top results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a DataFrame, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If include='all' is provided as an option, the result will include a union of attributes of each type.

The include and exclude parameters can be used to limit which columns in a DataFrame are analyzed for the output. The parameters are ignored when analyzing a Series.

### Examples

Describing a numeric Series.

```python
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count   4
unique  3
top     a
top freq 2
dtype: object
```
Describing a timestamp Series.

```python
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
```

```python
>>> s.describe(datetime_is_numeric=True)
count 3
mean 2006-09-01 08:00:00
min 2000-01-01 00:00:00
25% 2004-12-31 12:00:00
50% 2010-01-01 00:00:00
75% 2010-01-01 00:00:00
max 2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({
...     'categorical': pd.Categorical(['d', 'e', 'f']),
...     'numeric': [1, 2, 3],
...     'object': ['a', 'b', 'c']
... })
```

```python
>>> df.describe()
categorical numeric
count 3 3.0
unique 3 NaN 3
top f NaN a
freq 1 NaN 1
mean NaN 2.0 NaN
std 1.0 NaN
min NaN 1.0 NaN
25% NaN 1.5 NaN
50% NaN 2.0 NaN
75% NaN 2.5 NaN
max NaN 3.0 NaN
```

Describing all columns of a DataFrame regardless of data type.

```python
>>> df.describe(include='all')
categorical numeric object
count 3 3.0 3
unique 3 NaN 3
top f NaN a
freq 1 NaN 1
mean NaN 2.0 NaN
std NaN 1.0 NaN
min NaN 1.0 NaN
25% NaN 1.5 NaN
50% NaN 2.0 NaN
75% NaN 2.5 NaN
max NaN 3.0 NaN
```

Describing a column from a DataFrame by accessing it as an attribute.

```python
>>> df.numeric.describe()
count 3.0
```

(continues on next page)
Including only numeric columns in a DataFrame description.

```python
>>> df.describe(include=[np.number])
numeric
   count   mean   std   min   25%   50%   75%   max
numeric  3.0    2.0   1.0   1.0   1.5   2.0   2.5   3.0
Name: numeric, dtype: float64
```

Including only string columns in a DataFrame description.

```python
>>> df.describe(include=[object])
object
   count   unique    top    freq
object  3        3      a       1
```

Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
categorical
   count   unique   top    freq
categorical  3        3      d       1
```

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])
categorical object
   count 3  3
   unique 3  3
   top   f  a
   freq  1  1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[object])
categorical numeric
   count 3  3
   unique 3  3
   top   f  a
   freq  1  1
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>unique</td>
<td>3</td>
<td>NaN</td>
</tr>
<tr>
<td>top</td>
<td>f</td>
<td>NaN</td>
</tr>
<tr>
<td>freq</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>mean</td>
<td>NaN</td>
<td>2.0</td>
</tr>
<tr>
<td>std</td>
<td>NaN</td>
<td>1.0</td>
</tr>
<tr>
<td>min</td>
<td>NaN</td>
<td>1.0</td>
</tr>
<tr>
<td>25%</td>
<td>NaN</td>
<td>1.5</td>
</tr>
<tr>
<td>50%</td>
<td>NaN</td>
<td>2.0</td>
</tr>
<tr>
<td>75%</td>
<td>NaN</td>
<td>2.5</td>
</tr>
<tr>
<td>max</td>
<td>NaN</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**pandas.Series.diff**

**Series.diff(periods=1)**

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the Series (default is element in previous row).

**Parameters**

**periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.

**Returns**

**Series** First differences of the Series.

**See also:**

* **Series.pct_change** Percent change over given number of periods.
* **Series.shift** Shift index by desired number of periods with an optional time freq.
* **DataFrame.diff** First discrete difference of object.

**Notes**

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in Series, however dtype of the result is always float64.

**Examples**

Difference with previous row

```
>>> s = pd.Series([1, 1, 2, 3, 5, 8])
>>> s.diff()
0    NaN
1     0.0
2     1.0
3     1.0
4     2.0
```

(continues on next page)
Difference with 3rd previous row

```python
>>> s.diff(periods=3)
0   NaN
1   NaN
2   NaN
3     2.0
4     4.0
5     6.0
dtype: float64
```

Difference with following row

```python
>>> s.diff(periods=-1)
0    0.0
1   -1.0
2   -1.0
3   -2.0
4   -3.0
5     NaN
dtype: float64
```

Overflow in input dtype

```python
>>> s = pd.Series([1, 0], dtype=np.uint8)
>>> s.diff()
0   NaN
1  255.0
dtype: float64
```

### pandas.Series.div

**Series.div** *(other, level=None, fill_value=None, axis=0)*

Return Floating division of series and other, element-wise (binary operator `truediv`).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**
Series The result of the operation.

See also:

**Series.rtruediv** Reverse of the Floating division operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b   NaN
d    1.0
e   NaN
dtype: float64
>>> a.divide(b, fill_value=0)
a    1.0
b     Inf
c     Inf
d    0.0
e     NaN
dtype: float64
```

**pandas.Series.divide**

*Series.divide*(other, level=None, fill_value=None, axis=0)

Return Floating division of series and other, element-wise (binary operator `truediv`).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

Series The result of the operation.

See also:
Series.rtruediv  Reverse of the Floating division operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.divide(b, fill_value=0)
a    1.0
b    inf
c    inf
d    0.0
e    NaN
dtype: float64
```

pandas.Series.divmod

Series.divmod(other, level=None, fill_value=None, axis=0)

Return Integer division and modulo of series and other, element-wise (binary operator divmod).

Equivalent to divmod(series, other), but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

2-Tuple of Series The result of the operation.

See also:

Series.rdivmod  Reverse of the Integer division and modulo operator, see Python documentation for more details.
Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b     NaN
d    1.0
e    NaN
dtype: float64

>>> a.divmod(b, fill_value=0)
(a 1.0
b NaN
c NaN
d 0.0
e NaN
dtype: float64,
a 0.0
b NaN
c NaN
d 0.0
e NaN
dtype: float64)
```

**pandas.Series.dot**

*Series.dot*(other)

Compute the dot product between the Series and the columns of other.

This method computes the dot product between the Series and another one, or the Series and each columns of a DataFrame, or the Series and each columns of an array.

It can also be called using *self @ other* in Python >= 3.5.

**Parameters**

- **other** [Series, DataFrame or array-like] The other object to compute the dot product with its columns.

**Returns**

- **scalar, Series or numpy.ndarray** Return the dot product of the Series and other if other is a Series, the Series of the dot product of Series and each rows of other if other is a DataFrame or a numpy.ndarray between the Series and each columns of the numpy array.

**See also:**

- **DataFrame.dot** Compute the matrix product with the DataFrame.
Series.mul Multiplication of series and other, element-wise.

Notes
The Series and other has to share the same index if other is a Series or a DataFrame.

Examples
```python
>>> s = pd.Series([0, 1, 2, 3])
>>> other = pd.Series([-1, 2, -3, 4])
>>> s.dot(other)
8
>>> s @ other
8
>>> df = pd.DataFrame([[0, 1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(df)
0    24
1    14
dtype: int64
>>> arr = np.array([[0, 1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(arr)
array([24, 14])
```

pandas.Series.drop

Series.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters
- **labels** [single label or list-like] Index labels to drop.
- **axis** [0, default 0] Redundant for application on Series.
- **index** [single label or list-like] Redundant for application on Series, but 'index' can be used instead of 'labels'.
- **columns** [single label or list-like] No change is made to the Series; use 'index' or 'labels' instead.
- **level** [int or level name, optional] For MultiIndex, level for which the labels will be removed.
- **inplace** [bool, default False] If True, do operation inplace and return None.
- **errors** ['ignore', 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns
- **Series or None** Series with specified index labels removed or None if inplace=True.

3.3. Series 1259
**Raises**

**KeyError** If none of the labels are found in the index.

**See also:**

- **Series.reindex** Return only specified index labels of Series.
- **Series.dropna** Return series without null values.
- **Series.drop_duplicates** Return Series with duplicate values removed.
- **DataFrame.drop** Drop specified labels from rows or columns.

**Examples**

```python
>>> s = pd.Series(data=np.arange(3), index=['A', 'B', 'C'])
>>> s
A 0
B 1
C 2
dtype: int64

Drop labels B en C

>>> s.drop(labels=['B', 'C'])
A 0
dtype: int64
```

Drop 2nd level label in MultiIndex Series

```python
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                     ['speed', 'weight', 'length']],
...                     codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]],
...                     names=['species', 'measure'])
>>> s = pd.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...            index=midx)
>>> s
lama speed  45.0
    weight  200.0
    length  1.2
cow  speed  30.0
    weight  250.0
    length  1.5
falcon speed  320.0
    weight  1.0
    length  0.3
dtype: float64

>>> s.drop(labels='weight', level=1)
lama  speed  45.0
      length  1.2
cow   speed  30.0
      length  1.5
falcon speed  320.0
```

(continues on next page)
pandas.Series.drop_duplicates

Series.drop_duplicates(keep='first', inplace=False)
Return Series with duplicate values removed.

Parameters

keep [{'first', 'last', False}, default 'first'] Method to handle dropping duplicates:
• ‘first’ : Drop duplicates except for the first occurrence.
• ‘last’ : Drop duplicates except for the last occurrence.
• False : Drop all duplicates.

inplace [bool, default False] If True, performs operation inplace and returns None.

Returns
Series or None Series with duplicates dropped or None if inplace=True.

See also:

Index.drop_duplicates Equivalent method on Index.
DataFrame.drop_duplicates Equivalent method on DataFrame.
Series.duplicated Related method on Series, indicating duplicate Series values.

Examples

Generate a Series with duplicated entries.
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
    ...                  name='animal')
>>> s
0  lama
1  cow
2  lama
3  beetle
4  lama
5  hippo
Name: animal, dtype: object

With the ‘keep’ parameter, the selection behaviour of duplicated values can be changed. The value ‘first’ keeps the first occurrence for each set of duplicated entries. The default value of keep is ‘first’.

>>> s.drop_duplicates()
0  lama
1  cow
3  beetle
5  hippo
Name: animal, dtype: object
The value ‘last’ for parameter ‘keep’ keeps the last occurrence for each set of duplicated entries.

```python
>>> s.drop_duplicates(keep='last')
1 cow
3 beetle
4 lama
5 hippo
Name: animal, dtype: object
```

The value False for parameter ‘keep’ discards all sets of duplicated entries. Setting the value of ‘inplace’ to True performs the operation inplace and returns None.

```python
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s
1 cow
3 beetle
5 hippo
Name: animal, dtype: object
```

**pandas.Series.droplevel**

Series.droplevel(level, axis=0)

Return Series/DataFrame with requested index / column level(s) removed.

**Parameters**

- **level** [int, str, or list-like] If a string is given, must be the name of a level. If list-like, elements must be names or positional indexes of levels.
- **axis** [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the level(s) is removed:
  - 0 or ‘index’: remove level(s) in column.
  - 1 or ‘columns’: remove level(s) in row.

**Returns**

Series/DataFrame Series/DataFrame with requested index / column level(s) removed.

**Examples**

```python
>>> df = pd.DataFrame([...
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])

>>> df.columns = pd.MultiIndex.from_tuples([...
...     ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])

>>> df
  level_1  c  d
  level_2  e  f
```

(continues on next page)
pandas.Series.dropna

Series.dropna(
    axis=0, inplace=False, how=None
)  
Return a new Series with missing values removed.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or ‘index’}, default 0] There is only one axis to drop values from.

inplace [bool, default False] If True, do operation inplace and return None.


Returns

Series or None  Series with NA entries dropped from it or None if inplace=True.

See also:

Series.isna  Indicate missing values.

Series.notna  Indicate existing (non-missing) values.

Series.fillna  Replace missing values.

DataFrame.dropna  Drop rows or columns which contain NA values.

Index.dropna  Drop missing indices.
Examples

```python
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2   NaN
dtype: float64

Drop NA values from a Series.

```python
code
```n
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```n

Keep the Series with valid entries in the same variable.

```python
code
```n
```n
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```n

Empty strings are not considered NA values. None is considered an NA value.

```python
code
```n
```n
>>> ser = pd.Series([np.nan, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0     NaN
1      2
2     NaT
3
4    None
5    I stay
dtype: object
>>> ser.dropna()
1      2
3
5    I stay
dtype: object
```n

Examples

```python
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2   NaN
dtype: float64

Drop NA values from a Series.

```python
code
```n
```n
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```n

Keep the Series with valid entries in the same variable.

```python
code
```n
```n
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```n

Empty strings are not considered NA values. None is considered an NA value.
pandas.Series.dt

Series.dt()
Accessor object for datetimelike properties of the Series values.

Examples

```python
>>> seconds_series = pd.Series(pd.date_range("2000-01-01", periods=3, freq="s"))
>>> seconds_series
0 2000-01-01 00:00:00
1 2000-01-01 00:00:01
2 2000-01-01 00:00:02
dtype: datetime64[ns]
>>> seconds_series.dt.second
0 0
1 1
2 2
dtype: int64
```

```python
>>> hours_series = pd.Series(pd.date_range("2000-01-01", periods=3, freq="h"))
>>> hours_series
0 2000-01-01 00:00:00
1 2000-01-01 01:00:00
2 2000-01-01 02:00:00
dtype: datetime64[ns]
>>> hours_series.dt.hour
0 0
1 1
2 2
dtype: int64
```

```python
>>> quarters_series = pd.Series(pd.date_range("2000-01-01", periods=3, freq="q"))
>>> quarters_series
0 2000-03-31
1 2000-06-30
2 2000-09-30
dtype: datetime64[ns]
>>> quarters_series.dt.quarter
0 1
1 2
2 3
dtype: int64
```

Returns a Series indexed like the original Series. Raises TypeError if the Series does not contain datetimelike values.
pandas.Series.duplicated

Series.duplicated(keep='first')
Indicate duplicate Series values.
Duplicated values are indicated as True values in the resulting Series. Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.

Parameters

keep ([{'first', 'last', False}, default 'first'] Method to handle dropping duplicates:
  • 'first': Mark duplicates as True except for the first occurrence.
  • 'last': Mark duplicates as True except for the last occurrence.
  • False: Mark all duplicates as True.

Returns

Series[bool] Series indicating whether each value has occurred in the preceding values.

See also:

Index.duplicated Equivalent method on pandas.Index.
Dataframe.duplicated Equivalent method on pandas.DataFrame.
Series.drop_duplicates Remove duplicate values from Series.

Examples

By default, for each set of duplicated values, the first occurrence is set on False and all others on True:

```python
>>> animals = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> animals.duplicated()
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

which is equivalent to

```python
>>> animals.duplicated(keep='first')
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```python
>>> animals.duplicated(keep='last')
0     True
1    False
```

(continues on next page)
2 True
3 False
4 False
dtype: bool

By setting keep on False, all duplicates are True:

```python
>>> animals.duplicated(keep=False)
0   True
1   False
2   True
3   False
4   True
dtype: bool
```

### pandas.Series.eq

**Series.eq**(other, level=None, fill_value=None, axis=0)

Return Equal to of series and other, element-wise (binary operator `eq`).

Equivalent to `series == other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

**Series** The result of the operation.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a  
 a    1.0  
b    1.0  
c    1.0  
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b  
a    1.0  
b    NaN  
d    1.0  
```

(continues on next page)


```python
e NaN
dtype: float64
>>> a.eq(b, fill_value=0)
a   True
b   False
c   False
d   False
e   False
dtype: bool
```

**pandas.Series.equals**

Method: `Series.equals(other)`  

Test whether two objects contain the same elements.  

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal.

The row/column index do not need to have the same type, as long as the values are considered equal. Corresponding columns must be of the same dtype.

**Parameters**

- `other` [Series or DataFrame] The other Series or DataFrame to be compared with the first.

**Returns**

- `bool` True if all elements are the same in both objects, False otherwise.

**See also:**

- `Series.eq` Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.
- `DataFrame.eq` Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.
- `testing.assert_series_equal` Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.
- `testing.assert_frame_equal` Like assert_series_equal, but targets DataFrames.
- `numpy.array_equal` Return True if two arrays have the same shape and elements, False otherwise.

**Examples**

```python
>>> df = pd.DataFrame({1: [10], 2: [20]})
```

DataFrames `df` and `exactly_equal` have the same types and values for their elements and column labels, which will return True.
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
  1  2
0 10 20
>>> df.equals(exactly_equal)
True

DataFrames `df` and `different_column_type` have the same element types and values, but have different types for the column labels, which will still return True.

>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
  1.0  2.0
0 10.0 20.0
>>> df.equals(different_column_type)
True

DataFrames `df` and `different_data_type` have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
  1  2
0 10.0 20.0
>>> df.equals(different_data_type)
False

**pandas.Series.ewm**

Series.ewm(  
    com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True,  
    ignore_na=False, axis=0, times=None, method='single')

Provide exponentially weighted (EW) calculations.

Exactly one parameter: `com`, `span`, `halflife`, or `alpha` must be provided.

**Parameters**

- **com** [float, optional] Specify decay in terms of center of mass
  \[\alpha = \frac{1}{1 + com}, \text{for } com \geq 0.\]

- **span** [float, optional] Specify decay in terms of span
  \[\alpha = \frac{2}{span + 1}, \text{for } span \geq 1.\]

- **halflife** [float, str, timedelta, optional] Specify decay in terms of half-life
  \[\alpha = 1 - \exp(-\ln(2)/halflife), \text{for } halflife > 0.\]

  If `times` is specified, the time unit (str or timedelta) over which an observation decays to half its value. Only applicable to `mean()`, and halflife value will not apply to the other functions.

  New in version 1.1.0.

- **alpha** [float, optional] Specify smoothing factor \(\alpha\) directly
  \[0 < \alpha \leq 1.\]
**min_periods** [int, default 0] Minimum number of observations in window required to have a value; otherwise, result is np.nan.

**adjust** [bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

- When adjust=True (default), the EW function is calculated using weights \( w_i = (1 - \alpha)^i \). For example, the EW moving average of the series \([x_0, x_1, \ldots, x_t]\) would be:

\[
y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \ldots + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots + (1 - \alpha)^t}
\]

- When adjust=False, the exponentially weighted function is calculated recursively:

\[
y_0 = x_0 \\
y_t = (1 - \alpha)y_{t-1} + \alpha x_t,
\]

**ignore_na** [bool, default False] Ignore missing values when calculating weights.

- When ignore_na=False (default), weights are based on absolute positions. For example, the weights of \(x_0\) and \(x_2\) used in calculating the final weighted average of \([x_0, None, x_2]\) are \((1 - \alpha)^2\) and 1 if adjust=True, and \((1 - \alpha)^2\) and \(\alpha\) if adjust=False.

- When ignore_na=True, weights are based on relative positions. For example, the weights of \(x_0\) and \(x_2\) used in calculating the final weighted average of \([x_0, None, x_2]\) are \(1 - \alpha\) and 1 if adjust=True, and \(1 - \alpha\) and \(\alpha\) if adjust=False.

**axis** [[0, 1], default 0] If 0 or 'index', calculate across the rows.

If 1 or 'columns', calculate across the columns.

**times** [str, np.ndarray, Series, default None] New in version 1.1.0.

Only applicable to mean().

Times corresponding to the observations. Must be monotonically increasing and datetime64[ns] dtype.

If 1-D array like, a sequence with the same shape as the observations.

Deprecated since version 1.4.0: If str, the name of the column in the DataFrame representing the times.

**method** [str {'single', 'table'}, default 'single'] New in version 1.4.0.

Execute the rolling operation per single column or row ("single") or over the entire object ("table").

This argument is only implemented when specifying engine='numba' in the method call.

Only applicable to mean()

Returns

ExponentialMovingWindow subclass

See also:

- **rolling** Provides rolling window calculations.
- **expanding** Provides expanding transformations.
Notes

See *Windowing Operations* for further usage details and examples.

Examples

```python
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
    B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

>>> df.ewm(com=0.5).mean()
    B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

>>> df.ewm(alpha=2 / 3).mean()
    B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

adjust

>>> df.ewm(com=0.5, adjust=True).mean()
    B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

>>> df.ewm(com=0.5, adjust=False).mean()
    B
0  0.000000
1  0.666667
2  1.555556
3  1.555556
4  3.650794

ignore_na

>>> df.ewm(com=0.5, ignore_na=True).mean()
    B
0  0.000000
```

(continues on next page)
1 0.750000
2 1.615385
3 1.615385
4 3.225000

```python
>>> df.ewm(com=0.5, ignore_na=False).mean()
          B
0 0.000000
1 0.750000
2 1.615385
3 1.615385
4 3.670213
```

**times**

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```python
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
          B
0 0.000000
1 0.585786
2 1.523889
3 1.523889
4 3.233686
```

**pandas.Series.expanding**

`Series.expanding(min_periods=1, center=None, axis=0, method='single')` provides expanding window calculations.

**Parameters**

- `min_periods` [int, default 1] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`
- `center` [bool, default False] If False, set the window labels as the right edge of the window index.
  - If True, set the window labels as the center of the window index.
  
  Deprecated since version 1.1.0.
- `axis` [int or str, default 0] If 0 or 'index', roll across the rows.
  - If 1 or 'columns', roll across the columns.
- `method` [str {'single', 'table'}, default 'single'] Execute the rolling operation per single column or row ('single') or over the entire object ('table').
  
  This argument is only implemented when specifying `engine='numba'` in the method call.

**Returns**

- `Expanding subclass`
See also:

rolling Provides rolling window calculations.

ewm Provides exponential weighted functions.

Notes

See Windowing Operations for further usage details and examples.

Examples

```python
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
```

```python
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

min_periods

Expanding sum with 1 vs 3 observations needed to calculate a value.

```python
>>> df.expanding(1).sum()
   B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
```

```python
>>> df.expanding(3).sum()
   B
0  NaN
1  NaN
2  3.0
3  3.0
4  7.0
```

pandas.Series.explode

Series.explode( ignore_index=False )

Transform each element of a list-like to a row.

New in version 0.25.0.

Parameters

   ignore_index [bool, default False] If True, the resulting index will be labeled 0, 1, ..., n
                 - 1.

      New in version 1.1.0.
Returns

**Series** Exploded lists to rows; index will be duplicated for these rows.

See also:

*Series.str.split* Split string values on specified separator.

*Series.unstack* Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.

*DataFrame.melt* Unpivot a DataFrame from wide format to long format.

*DataFrame.explode* Explode a DataFrame from list-like columns to long format.

Notes

This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of elements in the output will be non-deterministic when exploding sets.

Reference *the user guide* for more examples.

Examples

```python
>>> s = pd.Series([[1, 2, 3], 'foo', [], [3, 4]])
>>> s
0   [1, 2, 3]
1     foo
2     []
3     [3, 4]
dtype: object

>>> s.explode()
0   1
0   2
0   3
1     foo
2    NaN
3     3
3     4
dtype: object
```

**pandas.Series.factorize**

*Series.factorize*(sort=False, na_sentinel=-1)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function *pandas.factorize()*, and as a method *Series.factorize()* and *Index.factorize()*.

Parameters

- **sort** [bool, default False] Sort *uniques* and shuffle *codes* to maintain the relationship.
**na_sentinel**  [int or None, default -1] Value to mark “not found”. If None, will not drop the NaN from the unique of the values.

Changed in version 1.1.2.

**Returns**

- **codes** [ndarray] An integer ndarray that’s an indexer into `uniques`. `uniques.take(codes)` will have the same values as `values`.
- **uniques** [ndarray, Index, or Categorical] The unique valid values. When `values` is Categorical, `uniques` is a Categorical. When `values` is some other pandas object, an `Index` is returned. Otherwise, a 1-D ndarray is returned.

**Note:** Even if there’s a missing value in `values`, `uniques` will not contain an entry for it.

**See also:**
- **cut** Discretize continuous-valued array.
- **unique** Find the unique value in an array.

**Notes**

Reference *the user guide* for more examples.

**Examples**

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> codes
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the `uniques` will be sorted, and `codes` will be shuffled so that the relationship is the maintained.

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> codes
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in `codes` with `na_sentinel` (-1 by default). Note that missing values are never included in `uniques`.

```python
>>> codes, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> codes
array([ 0, -1, 1, 2, 0])
```
Thus far, we’ve only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of `uniques` will differ. For Categoricals, a `Categorical` is returned.

```python
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
['a', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Notice that 'b' is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```python
>>> cat = pd.Series(['a', 'a', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype=object)
```

If NaN is in the values, and we want to include NaN in the uniques of the values, it can be achieved by setting `na_sentinel=None`.

```python
>>> values = np.array([1, 2, 1, np.nan])
>>> codes, uniques = pd.factorize(values)  # default: na_sentinel=-1
>>> codes
array([ 0, 1, 0, -1])
>>> uniques
array([ 1., 2.])
>>> codes, uniques = pd.factorize(values, na_sentinel=None)
>>> codes
array([0, 1, 0, 2])
>>> uniques
array([ 1., 2., nan])
```

### pandas.Series.ffill

**Series.ffill**(axis=None, inplace=False, limit=None, downcast=None)

Synonym for `DataFrame.fillna()` with method='ffill'.

**Returns**

Series/DataFrame or None  Object with missing values filled or None if inplace=True.
pandas.Series.fillna

Series.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)
Fill NA/Nan values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a
dict/Series/DataFrame of values specifying which value to use for each index (for a
Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not
be filled. This value cannot be a list.

holes in reindexed Series pad / fill: propagate last valid observation forward to next
valid backfill / bfill: use next valid observation to fill gap.

axis [{0 or ‘index’}] Axis along which to fill missing values.

inplace [bool, default False] If True, fill in-place. Note: this will modify any other views
on this object (e.g., a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive
NaNs values to forward/backward fill. In other words, if there is a gap with more than
this number of consecutive NaNs, it will only be partially filled. If method is not
specified, this is the maximum number of entries along the entire axis where NaNs
will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] A dict of item->dtype of what to downcast if possible, or
the string ‘infer’ which will try to downcast to an appropriate equal type (e.g. float64
to int64 if possible).

Returns

Series or None Object with missing values filled or None if inplace=True.

See also:

interpolate Fill NaN values using interpolation.
reindex Conform object to new index.
asfreq Convert TimeSeries to specified frequency.

Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                     [3, 4, np.nan, 1],
...                     [np.nan, np.nan, np.nan, np.nan],
...                     [np.nan, 3, np.nan, 4]],
...                     columns=list("ABCD"))
>>> df
   A  B   C  D
0  NaN 2.0   NaN 0.0
1  3.0 4.0  NaN  1.0
2  NaN NaN  NaN NaN
3  NaN 3.0  NaN  4.0
```
Replace all NaN elements with 0s.
df.fillna(0)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

We can also propagate non-null values forward or backward.

df.fillna(method="ffill")

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>2.0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>4.0</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>3.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Replace all NaN elements in column ‘A’, ‘B’, ‘C’, and ‘D’, with 0, 1, 2, and 3 respectively.

values = {"A": 0, "B": 1, "C": 2, "D": 3}
df.fillna(value=values)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Only replace the first NaN element.

df.fillna(value=values, limit=1)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>1.0</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>3.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

When filling using a DataFrame, replacement happens along the same column names and same indices

df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
df.fillna(df2)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Note that column D is not affected since it is not present in df2.
pandas.Series.filter

Series.filter(items=None, like=None, regex=None, axis=None)

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

- **items** [list-like] Keep labels from axis which are in items.
- **like** [str] Keep labels from axis for which “like in label == True”.
- **regex** [str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.
- **axis** [[0 or ‘index’, 1 or ‘columns’, None], default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame.

Returns

same type as input object

See also:

DataFrame.loc Access a group of rows and columns by label(s) or a boolean array.

Notes

The items, like, and regex parameters are enforced to be mutually exclusive. axis defaults to the info axis that is used when indexing with [].

Examples

```python
>>> df = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
   one  two  three
mouse  1   2    3
rabbit 4   5    6

>>> # select columns by name
>>> df.filter(items=['one', 'three'])
   one  three
mouse 1   3
rabbit 4   6

>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
   one  three
mouse 1   3
rabbit 4   6
```
# select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
     one  two  three
rabbit  4   5    6

**pandas.Series.first**

*Series.first(offset)*

Select initial periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can select the first few rows based on a date offset.

**Parameters**

- **offset** [str, DateOffset or dateutil.relativedelta] The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

**Returns**

- **Series or DataFrame** A subset of the caller.

**Raises**

- **TypeError** If the index is not a `DatetimeIndex`

**See also:**

- **last** Select final periods of time series based on a date offset.
- **at_time** Select values at a particular time of the day.
- **between_time** Select values between particular times of the day.

**Examples**

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
               A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4

Get the rows for the first 3 days:

```python
>>> ts.first('3D')
               A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.
pandas.Series.first_valid_index

Series.first_valid_index()
Return index for first non-NA value or None, if no non-NA value is found.

Returns
scalar [type of index]

Notes
If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

pandas.Series.floordiv

Series.floordiv(other, level=None, fill_value=None, axis=0)
Return integer division of series and other, element-wise (binary operator floordiv).
Equivalent to series // other, but with support to substitute a fill_value for missing data in either one
of the inputs.

Parameters
other [Series or scalar value]
fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values,
and any new element needed for successful Series alignment, with this value before
computation. If data in both corresponding Series locations is missing the result of
filling (at that location) will be missing.
level [int or name] Broadcast across a level, matching Index values on the passed Multi-
Index level.

Returns
Series The result of the operation.

See also:
Series.rfloordiv Reverse of the Integer division operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0
```

pandas.Series.ge

Series.ge(other, level=None, fill_value=None, axis=0)

Return Greater than or equal to of series and other, element-wise (binary operator ge).

Equivalent to series >= other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

Examples

```python
>>> a = pd.Series([1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    NaN
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
f    1.0
dtype: float64
>>> a.ge(b, fill_value=0)
```

(continues on next page)
pandas.Series.get

Series.get(key, default=None)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

Examples

```python
>>> df = pd.DataFrame(
    ... [24.3, 75.7, "high"],
    ... [31, 87.8, "high"],
    ... [22, 71.6, "medium"],
    ... [35, 95, "medium"],
    ... columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
    ... index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
    ... )

>>> df
   temp_celsius  temp_fahrenheit  windspeed
2014-02-12  24.3                75.7 high
2014-02-13  31.0                87.8 high
2014-02-14  22.0                71.6 medium
2014-02-15  35.0                95.0 medium

>>> df.get(["temp_celsius", "windspeed"])
   temp_celsius windspeed
2014-02-12  24.3    high
2014-02-13  31.0    high
2014-02-14  22.0   medium
2014-02-15  35.0   medium
```

If the key isn't found, the default value will be used.
pandas.Series.groupby

**Parameters**

- **by** [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If by is a function, it’s called on each value of the object’s index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series’ values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in self. Notice that a tuple is interpreted as a (single) key.

- **axis** [0 or ‘index’, 1 or ‘columns’, default 0] Split along rows (0) or columns (1).

- **level** [int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

- **as_index** [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively “SQL-style” grouped output.

- **sort** [bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

- **group_keys** [bool, default True] When calling apply, add group keys to index to identify pieces.

- **squeeze** [bool, default False] Reduce the dimensionality of the return type if possible, otherwise return a consistent type. Deprecated since version 1.1.0.

- **observed** [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

- **dropna** [bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups. New in version 1.1.0.

**Returns**

- **SeriesGroupBy** Returns a groupby object that contains information about the groups.

**See also:**

- **resample** Convenience method for frequency conversion and resampling of time series.
Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```python
>>> ser = pd.Series([390., 350., 30., 20.],
                  index=['Falcon', 'Falcon', 'Parrot', 'Parrot'], name="Max Speed")
>>> ser
Falcon    390.0
Falcon    350.0
Parrot    30.0
Parrot    20.0
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(["a", "b", "a", "b"]).mean()
a    210.0
b   185.0
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(level=0).mean()
Falcon    370.0
Parrot    25.0
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(ser > 100).mean()
Max Speed
False    25.0
True    370.0
Name: Max Speed, dtype: float64
```

Grouping by Indexes

We can groupby different levels of a hierarchical index using the `level` parameter:

```python
>>> arrays = [["Falcon", "Falcon", "Parrot", "Parrot"],
             ["Captive", "Wild", "Captive", "Wild"]]
>>> index = pd.MultiIndex.from_arrays(arrays, names=("Animal", "Type"))
>>> ser = pd.Series([390., 350., 30., 20.], index=index, name="Max Speed")
>>> ser
Animal  Type
Falcon  Captive    390.0
      Wild         350.0
Parrot  Captive    30.0
      Wild         20.0
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(level=0).mean()
Animal
Falcon    370.0
Parrot    25.0
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(level="Type").mean()
Type
Captive  210.0
```
Wild  185.0
Name: Max Speed, dtype: float64

We can also choose to include NA in group keys or not by defining dropna parameter, the default setting is True.

```python
>>> ser = pd.Series([1, 2, 3, 3], index=["a", 'a', 'b', np.nan])
>>> ser.groupby(level=0).sum()
a  3  
b  3  
dtype: int64
```

```python
>>> ser.groupby(level=0, dropna=False).sum()
a  3  
b  3  
NaN  3  
dtype: int64
```

```python
arrays = ['Falcon', 'Falcon', 'Parrot', 'Parrot']
>>> ser = pd.Series([390., 350., 30., 20.], index=arrays, name="Max Speed")
>>> ser.groupby(['a', 'b', 'a', np.nan]).mean()
a  210.0  
b  350.0  
Name: Max Speed, dtype: float64
```

```python
>>> ser.groupby(['a', 'b', 'a', np.nan], dropna=False).mean()
a  210.0  
b  350.0  
NaN  20.0  
Name: Max Speed, dtype: float64
```

### pandas.Series.gt

**Series.gt**(other, level=None, fill_value=None, axis=0)

Return Greater than of series and other, element-wise (binary operator gt).

Equivalent to series > other, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **Series** The result of the operation.
Examples

```python
>>> a = pd.Series([1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
d    NaN
e    1.0
b    1.0
c    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
d    NaN
f    1.0
c    2.0
b    1.0
dtype: float64
>>> a.gt(b, fill_value=0)
a    True
d    False
c    False
e    True
f    False
dtype: bool
```

**pandas.Series.head**

Series.head(n=5)

Return the first $n$ rows.

This function returns the first $n$ rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of $n$, this function returns all rows except the last $n$ rows, equivalent to df[:\:-n].

**Parameters**

- **n** [int, default 5] Number of rows to select.

**Returns**

- **same type as caller** The first $n$ rows of the caller object.

**See also:**

- **DataFrame.tail** Returns the last $n$ rows.
Examples

```python
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion', 'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
animal
0  alligator
1    bee
2   falcon
3    lion
4  monkey
5  parrot
6    shark
7   whale
8   zebra
```

Viewing the first 5 lines

```python
>>> df.head()
animal
0  alligator
1    bee
2   falcon
3    lion
4  monkey
```

Viewing the first $n$ lines (three in this case)

```python
>>> df.head(3)
animal
0  alligator
1    bee
2   falcon
```

For negative values of $n$

```python
>>> df.head(-3)
animal
0  alligator
1    bee
2   falcon
3    lion
4  monkey
5  parrot
```
pandas.Series.hist

Series.hist(by=None, ax=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, figsize=None, bins=10, backend=None, legend=False, **kwargs)

Draw histogram of the input series using matplotlib.

Parameters

by [object, optional] If passed, then used to form histograms for separate groups.

ax [matplotlib axis object] If not passed, uses gca().

grid [bool, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels.

figsize [tuple, default None] Figure size in inches by default.

bins [int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend.

legend [bool, default False] Whether to show the legend.

**kwargs To be passed to the actual plotting function.

Returns

matplotlib.AxesSubplot A histogram plot.

See also:

matplotlib.axes.Axes.hist Plot a histogram using matplotlib.

pandas.Series.idxmax

Series.idxmax(axis=0, skipna=True, *args, **kwargs)

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

Parameters

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

skipna [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.
**pandas.Series.idxmax**

### Syntax

`Series.idxmax(axis=0, skipna=True, *args, **kwargs)`

### Description

Return the row label of the minimum value. If multiple values equal the minimum, the first row label with that value is returned.

### Parameters

- **axis** [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

### Examples

```python
g = pd.DataFrame(data=[1, None, 4, 3, 4], index=['A', 'B', 'C', 'D', 'E'])
g

A  1.0
B NaN
C  4.0
D  3.0
E  4.0
dtype: float64

>>> g.idxmin()  
'B'
```

If `skipna` is False and there is an NA value in the data, the function returns `nan`.

```python
>>> g.idxmin(skipna=False)  
nan
```

---

*args, **kwargs Additional arguments and keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Index** Label of the maximum value.

**Raises**

- **ValueError** If the Series is empty.

See also:

- `numpy.argmax` Return indices of the maximum values along the given axis.
- `DataFrame.idxmax` Return index of first occurrence of maximum over requested axis.
- `Series.idxmin` Return index label of the first occurrence of minimum of values.

**Notes**

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.
skipna [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

*args, **kwargs Additional arguments and keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Index Label of the minimum value.

Raises

ValueError If the Series is empty.

See also:

numpy.argmin Return indices of the minimum values along the given axis.

DataFrame.idxmin Return index of first occurrence of minimum over requested axis.

Series.idxmax Return index label of the first occurrence of maximum of values.

Notes

This method is the Series version of ndarray.argmin. This method returns the label of the minimum, while ndarray.argmin returns the position. To get the position, use series.values.argmin().

Examples

```python
>>> s = pd.Series(data=[1, None, 4, 1],
                 index=['A', 'B', 'C', 'D'])
>>> s
A   1.0
B   NaN
C   4.0
D   1.0
dtype: float64

>>> s.idxmin()
'A'
```

If skipna is False and there is an NA value in the data, the function returns nan.

```python
>>> s.idxmin(skipna=False)
nan
```
**pandas.Series.infer_objects**

Series.infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

Returns

- **converted** [same type as input object]

See also:

- **to_datetime** Convert argument to datetime.
- **to_timedelta** Convert argument to timedelta.
- **to_numeric** Convert argument to numeric type.
- **convert_dtypes** Convert argument to best possible dtype.

**Examples**

```python
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
0  1
1  2
2  3

>>> df.dtypes
A  object
dtype: object

>>> df.infer_objects().dtypes
A  int64
dtype: object
```

**pandas.Series.info**

Series.info(\*\*kwargs)

Print a concise summary of a Series.

This method prints information about a Series including the index dtype, non-null values and memory usage.

New in version 1.4.0.

Parameters

- **data** [Series] Series to print information about.

- **verbose** [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.
buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

memory_usage [bool, str, optional] Specifies whether total memory usage of the Series elements (including the index) should be displayed. By default, this follows the pandas.options.display.memory_usage setting.

True always show memory usage. False never shows memory usage. A value of ‘deep’ is equivalent to “True with deep introspection”. Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

show_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than pandas.options.display.max_info_rows and pandas.options.display.max_info_columns. A value of True always shows the counts, and False never shows the counts.

Returns

None This method prints a summary of a Series and returns None.

See also:

Series.describe Generate descriptive statistics of Series.

Series.memory_usage Memory usage of Series.

Examples

```python
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> s = pd.Series(text_values, index=int_values)
>>> s.info()
<class 'pandas.core.series.Series'>
Int64Index: 5 entries, 1 to 5
Series name: None
Non-Null Count  Dtype
-------------  -----  
5 non-null      object
dtypes: object(1)
memory usage: 80.0+ bytes
```

Prints a summary excluding information about its values:

```python
>>> s.info(verbosr=False)
<class 'pandas.core.series.Series'>
Int64Index: 5 entries, 1 to 5
dtypes: object(1)
memory usage: 80.0+ bytes
```

Pipe output of Series.info to buffer instead of sys.stdout, get buffer content and writes to a text file:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> import io
>>> buffer = io.StringIO()
>>> s.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...         encoding="utf-8") as f:
...     f.write(s)
```

The `memory_usage` parameter allows deep introspection mode, specially useful for big Series and fine-tune memory optimization:

```python
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> s = pd.Series(np.random.choice(['a', 'b', 'c'], 10 ** 6))
>>> s.info()
<class 'pandas.core.series.Series'>
RangeIndex: 1000000 entries, 0 to 999999
Series name: None
Non-Null Count   Dtype
1000000 non-null object
dtypes: object(1)
memory usage: 7.6+ MB

>>> s.info(memory_usage='deep')
<class 'pandas.core.series.Series'>
RangeIndex: 1000000 entries, 0 to 999999
Series name: None
Non-Null Count   Dtype
1000000 non-null object
dtypes: object(1)
memory usage: 55.3 MB
```

### pandas.Series.interpolate

`Series.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)`

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

**Parameters**

- **method** [str, default ‘linear’] Interpolation technique to use. One of:
  - ‘linear’: Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
  - ‘time’: Works on daily and higher resolution data to interpolate given length of interval.
  - ‘index’, ‘values’: use the actual numerical values of the index.
  - ‘pad’: Fill in NaNs using existing values.
• ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘spline’, ‘barycentric’, ‘polynomial’: Passed to scipy.interpolate.interp1d. These methods use the numerical values of the index. Both ‘polynomial’ and ‘spline’ require that you also specify an order (int), e.g. df.interpolate(method='polynomial', order=5).


• ‘from_derivatives’: Refers to scipy.interpolate.BPoly.from_derivatives which replaces ‘piecewise_polynomial’ interpolation method in scipy 0.18.

axis | [{0 or ‘index’, 1 or ‘columns’, None}], default None | Axis to interpolate along.

limit | [int, optional] | Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace | [bool, default False] | Update the data in place if possible.

limit_direction | [{‘forward’, ‘backward’, ‘both’}], Optional | Consecutive NaNs will be filled in this direction.

  If limit is specified:
  • If ‘method’ is ‘pad’ or ‘ffill’, ‘limit_direction’ must be ‘forward’.
  • If ‘method’ is ‘backfill’ or ‘bfill’, ‘limit_direction’ must be ‘backwards’.

  If ‘limit’ is not specified:
  • If ‘method’ is ‘backfill’ or ‘bfill’, the default is ‘backward’
  • else the default is ‘forward’

Changed in version 1.1.0: raises ValueError if limit_direction is ‘forward’ or ‘both’ and method is ‘backfill’ or ‘bfill’. raises ValueError if limit_direction is ‘backward’ or ‘both’ and method is ‘pad’ or ‘ffill’.

limit_area | [{None, ‘inside’, ‘outside’}], default None | If limit is specified, consecutive NaNs will be filled with this restriction.

  • None: No fill restriction.
  • ‘inside’: Only fill NaNs surrounded by valid values (interpolate).
  • ‘outside’: Only fill NaNs outside valid values (extrapolate).

downcast | [optional, ‘infer’ or None, defaults to None] | Downcast dtypes if possible.

**kwargs | [optional] | Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame or None | Returns the same object type as the caller, interpolated at some or all NaN values or None if inplace=True.

See also:

fillna | Fill missing values using different methods.

scipy.interpolate.Akima1DInterpolator | Piecewise cubic polynomials (Akima interpolator).

scipy.interpolate.BPoly.from_derivatives | Piecewise polynomial in the Bernstein basis.

scipy.interpolate.interp1d | Interpolate a 1-D function.

scipy.interpolate.KroghInterpolator | Interpolate polynomial (Krogh interpolator).
scipy.interpolate.PchipInterpolator  PCHIP 1-d monotonic cubic interpolation.
scipy.interpolate.CubicSpline  Cubic spline data interpolator.

Notes

The ‘krogh’, ‘piecewise_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in NaN in a Series via linear interpolation.

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0   0.0
1   1.0
2   NaN
3   3.0
dtype: float64
>>> s.interpolate()
0   0.0
1   1.0
2   2.0
3   3.0
dtype: float64
```

Filling in NaN in a Series by padding, but filling at most two consecutive NaN at a time.

```python
>>> s = pd.Series([np.nan, "single_one", np.nan, ...
...     "fill_two_more", np.nan, np.nan, np.nan, ...
...     4.71, np.nan])
>>> s
0   NaN
1   single_one
2   NaN
3   fill_two_more
4   NaN
5   NaN
6   NaN
7   4.71
8   NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0   NaN
1   single_one
2   single_one
3   fill_two_more
4   fill_two_more
5   fill_two_more
6   NaN
7   4.71
```

(continues on next page)
Filling in NaN in a Series via polynomial interpolation or splines: Both ‘polynomial’ and ‘spline’ methods require that you also specify an order (int).

```python
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column ‘a’ is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column ‘b’ remains NaN, because there is no entry before it to use for interpolation.

```python
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                     (np.nan, 2.0, np.nan, np.nan),
...                     (2.0, 3.0, np.nan, 9.0),
...                     (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
      a     b     c     d
0  0.0   NaN  -1.0   1.0
1  NaN   2.0   NaN   NaN
2  2.0   3.0   NaN   9.0
3  NaN   4.0  -4.0  16.0
```

```python
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
      a     b     c     d
0  0.0   NaN  -1.0   1.0
1  1.0   2.0  -2.0   5.0
2  2.0   3.0  -3.0   9.0
3  2.0   4.0  -4.0  16.0
```

Using polynomial interpolation.

```python
>>> df['d'].interpolate(method='polynomial', order=2)
0    1.0
1    4.0
2    9.0
3   16.0
Name: d, dtype: float64
```
pandas.Series.isin

Series.isin(values)

Whether elements in Series are contained in values.

Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of values exactly.

Parameters

values [set or list-like] The sequence of values to test. Passing in a single string will raise a TypeError. Instead, turn a single string into a list of one element.

Returns

Series Series of booleans indicating if each element is in values.

Raises

TypeError

• If values is a string

See also:

DataFrame.isin Equivalent method on DataFrame.

Examples

```python
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...                'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0    True
1    True
2    True
3   False
4    True
5   False
Name: animal, dtype: bool
```

To invert the boolean values, use the ~ operator:

```python
>>> ~s.isin(['cow', 'lama'])
0   False
1   False
2   False
3    True
4   False
5    True
Name: animal, dtype: bool
```

Passing a single string as s.isin('lama') will raise an error. Use a list of one element instead:

```python
>>> s.isin(['lama'])
0    True
1   False
(continues on next page)"
Strings and integers are distinct and are therefore not comparable:

```python
>>> pd.Series([1]).isin([1])
0    False
dtype: bool
```

```python
>>> pd.Series([1.1]).isin([1.1])
0    False
dtype: bool
```

### pandas.Series.isna

Series.isna() Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy. NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

**Returns**

Series Mask of bool values for each element in Series that indicates whether an element is an NA value.

**See also:**

- **Series.isnull** Alias of isna.
- **Series.notna** Boolean inverse of isna.
- **Series.dropna** Omit axes labels with missing values.
- **isna** Top-level isna.

**Examples**

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                         pd.Timestamp('1940-04-25')],
...                    name=['Alfred', 'Batman', ''],
...                    toy=[None, 'Batmobile', 'Joker']))
```

<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>born</th>
<th>name</th>
<th>toy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.0</td>
<td>NaT</td>
<td>Alfred</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>6.0</td>
<td>1939-05-27</td>
<td>Batman</td>
<td>Batmobile</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>1940-04-25</td>
<td>Joker</td>
<td></td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.isna()
   age  born  name  toy
0  False  True False  True
1  False False False False
2   True False False False

Show which entries in a Series are NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64

```python
>>> ser.isna()
0   False
1   False
2    True
dtype: bool

pandas.Series.isnull

Series.isnull()  
Series.isnull is an alias for Series.isna.  
Detect missing values.  
Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy. NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns  
Series  Mask of bool values for each element in Series that indicates whether an element is an NA value.

See also:  
Series.isnull  Alias of isna.  
Series.notna  Boolean inverse of isna.  
Series.dropna  Omit axes labels with missing values.  
isna  Top-level isna.
Examples

Show which entries in a DataFrame are NA.

```python
df = pd.DataFrame(dict(age=[5, 6, np.NaN],
                         pd.Timestamp('1940-04-25')],
                   name=['Alfred', 'Batman', ''],
                   toy=[None, 'Batmobile', 'Joker']))
```

```python
df
```
```
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>born</th>
<th>name</th>
<th>toy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.0</td>
<td>NaT</td>
<td>Alfred</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>6.0</td>
<td>1939-05-27</td>
<td>Batman</td>
<td>Batmobile</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>1940-04-25</td>
<td>Joker</td>
<td></td>
</tr>
</tbody>
</table>
```

```python
df.isna()
```
```
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>born</th>
<th>name</th>
<th>toy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
```

Show which entries in a Series are NA.

```python
ser = pd.Series([5, 6, np.NaN])
```

```python
ser
```
```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```

```python
ser.isna()
```
```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```

pandas.Series.item

Series.item()

Return the first element of the underlying data as a Python scalar.

Returns

scalar The first element of %(klass)s.

Raises

ValueError If the data is not length-1.
pandas.Series.items

Series.items()

Lazily iterate over (index, value) tuples.

This method returns an iterable tuple (index, value). This is convenient if you want to create a lazy iterator.

Returns

iterable Iterable of tuples containing the (index, value) pairs from a Series.

See also:

DataFrame.items Iterate over (column name, Series) pairs.

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

Examples

```python
>>> s = pd.Series(['A', 'B', 'C'])
>>> for index, value in s.items():
...    print(f'Index : {index}, Value : {value}')
Index : 0, Value : A
Index : 1, Value : B
Index : 2, Value : C
```

pandas.Series.iteritems

Series.iteritems()

Lazily iterate over (index, value) tuples.

This method returns an iterable tuple (index, value). This is convenient if you want to create a lazy iterator.

Returns

iterable Iterable of tuples containing the (index, value) pairs from a Series.

See also:

DataFrame.items Iterate over (column name, Series) pairs.

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

Examples

```python
>>> s = pd.Series(['A', 'B', 'C'])
>>> for index, value in s.items():
...    print(f'Index : {index}, Value : {value}')
Index : 0, Value : A
Index : 1, Value : B
Index : 2, Value : C
```
**pandas.Series.keys**

```python
Series.keys()
```

Return alias for index.

**Returns**

- **Index**: Index of the Series.

**pandas.Series.kurt**

```python
Series.kurt(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)
```

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

**Parameters**

- **axis** ([index (0)]) Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **kwargs** Additional keyword arguments to be passed to the function.

**Returns**

- **scalar or Series (if level specified)**

**pandas.Series.kurtosis**

```python
Series.kurtosis(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)
```

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

**Parameters**

- **axis** ([index (0)]) Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **kwargs** Additional keyword arguments to be passed to the function.

**Returns**

- **scalar or Series (if level specified)**
pandas.Series.last

Series.last(offset)
Select final periods of time series data based on a date offset.
For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters
offset [str, DateOffset, dateutil.relativedelta] The offset length of the data that will be selected. For instance, ‘3D’ will display all the rows having their index within the last 3 days.

Returns
Series or DataFrame A subset of the caller.

Raises
TypeError If the index is not a DatetimeIndex

See also:
first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)

Get the rows for the last 3 days:

```python
>>> ts.last('3D')
A
2018-04-13 3
2018-04-15 4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.
pandas.Series.last_valid_index

Series.last_valid_index()
Return index for last non-NA value or None, if no non-NA value is found.

Returns
scalar [type of index]

Notes
If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

pandas.Series.le

Series.le(other, level=None, fill_value=None, axis=0)
Return Less than or equal to of series and other, element-wise (binary operator le).
Equivalent to series <= other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters
other [Series or scalar value]
fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns
Series The result of the operation.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
f    1.0
```
pandas.Series.lt

Series.lt(other, level=None, fill_value=None, axis=0)

Return Less than of series and other, element-wise (binary operator lt).

Equivalent to series < other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters
- other [Series or scalar value]
- fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns
Series The result of the operation.

Examples

>>> a = pd.Series([1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
f    1.0
dtype: float64
>>> a.lt(b, fill_value=0)
pandas.Series.mad

Series.mad(axis=None, skipna=True, level=None)

Return the mean absolute deviation of the values over the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

Returns

scalar or Series (if level specified)

pandas.Series.map

Series.map(arg, na_action=None)

Map values of Series according to an input mapping or function.

Used for substituting each value in a Series with another value, that may be derived from a function, a dict or a Series.

Parameters

na_action [{None, ‘ignore’}, default None] If ‘ignore’, propagate NaN values, without passing them to the mapping correspondence.

Returns

Series Same index as caller.

See also:

Series.apply For applying more complex functions on a Series.
DataFrame.apply Apply a function row-/column-wise.
DataFrame.applymap Apply a function elementwise on a whole DataFrame.
Notes

When `arg` is a dictionary, values in Series that are not in the dictionary (as keys) are converted to NaN. However, if the dictionary is a dict subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than NaN.

Examples

```python
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])
>>> s
0   cat
1   dog
2  NaN
3  rabbit
dtype: object
```

Map accepts a dict or a Series. Values that are not found in the dict are converted to NaN, unless the dict has a default value (e.g. `defaultdict`):

```python
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0   kitten
1    puppy
2    NaN
3    NaN
dtype: object
```

It also accepts a function:

```python
>>> s.map('I am a {}.format')
0    I am a cat
1    I am a dog
2    I am a nan
3    I am a rabbit
dtype: object
```

To avoid applying the function to missing values (and keep them as NaN) `na_action='ignore'` can be used:

```python
>>> s.map('I am a {}.format', na_action='ignore')
0    I am a cat
1    I am a dog
2       NaN
3    I am a rabbit
dtype: object
```
pandas.Series.mask

Series.mask(cond, other=nan, inplace=False, axis=None, level=None, errors=NoDefault.no_default, try_cast=NoDefault.no_default)

Replace values where the condition is True.

Parameters

cond [bool Series/DataFrame, array-like, or callable] Where cond is False, keep the original value. Where True, replace with corresponding value from other. If cond is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

other [scalar, Series/DataFrame, or callable] Entries where cond is True are replaced with corresponding value from other. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

inplace [bool, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default 'raise'] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

• ‘raise’ : allow exceptions to be raised.
• ‘ignore’ : suppress exceptions. On error return original object.

try_cast [bool, default None] Try to cast the result back to the input type (if possible).

Deprecated since version 1.3.0: Manually cast back if necessary.

Returns

Same type as caller or None if inplace=True.

See also:

DataFrame.where() Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if cond is False the element is used; otherwise the corresponding element from the DataFrame other is used.

The signature for DataFrame.where() differs from numpy.where(). Roughly df1.where(m, df2) is equivalent to np.where(m, df1, df2).

For further details and examples see the mask documentation in indexing.
Examples

```python
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0   NaN
1   1.0
2   2.0
3   3.0
4   4.0
dtype: float64

>>> s.mask(s > 0)
0   0.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64

>>> s.where(s > 1, 10)
0  10
1  10
2   2
3   3
4   4
dtype: int64

>>> s.mask(s > 1, 10)
0   0
1   1
2  10
3  10
4  10
dtype: int64

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9

>>> m = df % 3 == 0
>>> df.where(m, -df)
   A   B
0  0  -1
1  2   3
2 -4  -5
3  6  -7
4 -8   9

>>> df.where(m, -df) == np.where(m, df, -df)
   A   B
0  True True
```

(continues on next page)
pandas.Series.max

Series.max(axis=None, skipna=True, level=None, numeric_only=None, **kwargs)

Return the maximum of the values over the requested axis.

If you want the index of the maximum, use idxmax. This is the equivalent of the numpy.ndarray method argmax.

Parameters

- axis [(index (0))] Axis for the function to be applied on.
- skipna [bool, default True] Exclude NA/null values when computing the result.
- level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

- Series.sum Return the sum.
- Series.min Return the minimum.
- Series.max Return the maximum.
- Series.idxmin Return the index of the minimum.
- Series.idxmax Return the index of the maximum.
- DataFrame.sum Return the sum over the requested axis.
- DataFrame.min Return the minimum over the requested axis.
- DataFrame.max Return the maximum over the requested axis.
- DataFrame.idxmin Return the index of the minimum over the requested axis.
- DataFrame.idxmax Return the index of the maximum over the requested axis.
Examples

```python
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']
...     names=['blooded', 'animal'])

>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)

>>> s
blooded animal
warm dog  4
    falcon  2
cold fish  0
    spider  8
Name: legs, dtype: int64

>>> s.max()
8
```

**pandas.Series.mean**

Series.mean(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return the mean of the values over the requested axis.

**Parameters**

- **axis**  
  [[index (0)]] Axis for the function to be applied on.

- **skipna**  
  [bool, default True] Exclude NA/null values when computing the result.

- **level**  
  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

- **numeric_only**  
  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

- **kwargs**  
  Additional keyword arguments to be passed to the function.

**Returns**

- scalar or Series (if level specified)

**pandas.Series.median**

Series.median(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return the median of the values over the requested axis.

**Parameters**

- **axis**  
  [[index (0)]] Axis for the function to be applied on.

- **skipna**  
  [bool, default True] Exclude NA/null values when computing the result.

- **level**  
  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

- **numeric_only**  
  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
**kwargs Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.memory_usage

Series.memory_usage(index=True, deep=False)

Return the memory usage of the Series.

The memory usage can optionally include the contribution of the index and of elements of object dtype.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the Series index.

deep [bool, default False] If True, introspect the data deeply by interrogating object dtypes for system-level memory consumption, and include it in the returned value.

Returns

int Bytes of memory consumed.

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of the array.

DataFrame.memory_usage Bytes consumed by a DataFrame.

Examples

```python
>>> s = pd.Series(range(3))
>>> s.memory_usage()
152

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```python
>>> s.memory_usage(index=False)
24
```

The memory footprint of object values is ignored by default:

```python
>>> s = pd.Series(["a", "b"])
>>> s.values
array(['a', 'b'], dtype=object)
>>> s.memory_usage()
144
>>> s.memory_usage(deep=True)
244
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.Series.min

Series.min(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return the minimum of the values over the requested axis.

If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

Parameters

axis [(index (0))] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.sum Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
def idx = pd.MultiIndex.from_arrays([
    ['warm', 'warm', 'cold', 'cold'],
    ['dog', 'falcon', 'fish', 'spider']],
    names=['blooded', 'animal'])
def s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>> s
blooded animal
dog 4
falcon 2
fish 0
```
spider 8
Name: legs, dtype: int64

>>> s.min()
0

### pandas.Series.mod

**Series.mod(other, level=None, fill_value=None, axis=0)**

Return Modulo of series and other, element-wise (binary operator mod).

Equivalent to `series % other`, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **Series** The result of the operation.

**See also:**

- **Series.rmod** Reverse of the Modulo operator, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d   NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b   NaN
d    1.0
e   NaN
dtype: float64
>>> a.mod(b, fill_value=0)
a    0.0
b   NaN
```
pandas.Series.mode

Series.mode(dropna=True)
Return the mode(s) of the Series.

The mode is the value that appears most often. There can be multiple modes.
Always returns Series even if only one value is returned.

Parameters

  dropna [bool, default True] Don’t consider counts of NaN/NaT.

Returns

  Series  Modes of the Series in sorted order.

pandas.Series.mul

Series.mul(other, level=None, fill_value=None, axis=0)
Return Multiplication of series and other, element-wise (binary operator mul).
Equivalent to series * other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

  other [Series or scalar value]

  fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.

  level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

  Series  The result of the operation.

See also:

  Series.rmul  Reverse of the Multiplication operator, see Python documentation for more details.
Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64

>>> a.multiply(b, fill_value=0)
a 1.0
b 0.0
c 0.0
d 0.0
e NaN
dtype: float64
```

**pandas.Series.multiply**

`Series.multiply(other, level=None, fill_value=None, axis=0)`

Return Multiplication of series and other, element-wise (binary operator `mul`).

Equivalent to `series * other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **Series** The result of the operation.

**See also:**

- `Series.rmul` Reverse of the Multiplication operator, see Python documentation for more details.
Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.multiply(b, fill_value=0)
a 1.0
b 0.0
c 0.0
d 0.0
e NaN
dtype: float64
```

`pandas.Series.ne`

`Series.ne(other, level=None, fill_value=None, axis=0)`

Return Not equal to of series and other, element-wise (binary operator `ne`).

Equivalent to `series != other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

Parameters

- `other` [Series or scalar value]
- `fill_value` [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- `level` [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns

- `Series` The result of the operation.
Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
```

```python
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
```

```python
>>> a.ne(b, fill_value=0)
a  False
b   True
c   True
d   True
e   True
dtype: bool
```

**pandas.Series.nlargest**

`Series.nlargest(n=5, keep='first')`

Return the largest `n` elements.

**Parameters**

- `n` [int, default 5] Return this many descending sorted values.
- `keep` [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of `n` elements:
  - `first` : return the first `n` occurrences in order of appearance.
  - `last` : return the last `n` occurrences in reverse order of appearance.
  - `all` : keep all occurrences. This can result in a Series of size larger than `n`.

**Returns**

- `Series` The `n` largest values in the Series, sorted in decreasing order.

**See also:**

- `Series.nsmallest` Get the `n` smallest elements.
- `Series.sort_values` Sort Series by values.
- `Series.head` Return the first `n` rows.
Notes

Faster than `.sort_values(ascending=False).head(n)` for small `n` relative to the size of the `Series` object.

Examples

```python
>>> countries_population = {"Italy": 59000000, "France": 65000000,
"Malta": 434000, "Maldives": 434000,
"Brunei": 434000, "Iceland": 337000,
"Nauru": 11300, "Tuvalu": 11300,
"Anguilla": 11300, "Montserrat": 5200}

>>> s = pd.Series(countries_population)

The `n` largest elements where `n=5` by default.

```python
>>> s.nlargest()
France  65000000
Italy   59000000
Malta   434000
Maldives 434000
Brunei  434000
Nauru   11300
Tuvalu  11300
Anguilla 11300
Montserrat 5200
dtype: int64
```

The `n` largest elements where `n=3`. Default `keep` value is `first` so Malta will be kept.

```python
>>> s.nlargest(3)
France  65000000
Italy   59000000
Malta   434000
Nauru   11300
Tuvalu  11300
Anguilla 11300
Montserrat 5200
dtype: int64
```

The `n` largest elements where `n=3` and keeping the last duplicates. Brunei will be kept since it is the last with value 434000 based on the index order.

```python
>>> s.nlargest(3, keep='last')
France  65000000
Italy   59000000
Brunei  434000
```
The $n$ largest elements where $n=3$ with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```python
>>> s.nlargest(3, keep='all')
France       65000000
Italy        59000000
Malta        4340000
Maldives     4340000
Brunei       4340000
dtype: int64
```

**pandas.Series.notna**

Series.notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

**Returns**

- **Series**: Mask of bool values for each element in Series that indicates whether an element is not an NA value.

**See also:**

- **Series.notnull**: Alias of notna.
- **Series.isna**: Boolean inverse of notna.
- **Series.dropna**: Omit axes labels with missing values.
- **notna**: Top-level notna.

**Examples**

Show which entries in a DataFrame are not NA.

```python
>>> df = pd.DataFrame(dict(age=[5, 6, np.nan],
...                   born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                         pd.Timestamp('1940-04-25')],
...                   name=['Alfred', 'Batman', ''],
...                   toy=[None, 'Batmobile', 'Joker']))
>>> df
   age   born            name   toy
0   5.0  NaT        Alfred     None
1   6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25     Joker     None

```
Show which entries in a Series are not NA.

```python
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```python
>>> ser.notna()
0    True
1    True
2    False
dtype: bool
```

**pandas.Series.notnull**

`Series.notnull()`

Series.notnull is an alias for Series.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

**Returns**

- `Series` Mask of bool values for each element in Series that indicates whether an element is not an NA value.

**See also:**

- `Series.notnull` Alias of notna.
- `Series.isna` Boolean inverse of notna.
- `Series.dropna` Omit axes labels with missing values.
- `notna` Top-level notna.
Examples

Show which entries in a DataFrame are not NA.

```python
df = pd.DataFrame(dict(age=[5, 6, np.NaN],
          pd.Timestamp('1940-04-25')],
    name=['Alfred', 'Batman', ''],
    toy=[None, 'Batmobile', 'Joker']))
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>born</th>
<th>name</th>
<th>toy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.0</td>
<td>NaN</td>
<td>Alfred</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>6.0</td>
<td>1939-05-27</td>
<td>Batman</td>
<td>Batmobile</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>1940-04-25</td>
<td>Joker</td>
<td></td>
</tr>
</tbody>
</table>
```

```python
df.notna()
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>born</th>
<th>name</th>
<th>toy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
```

Show which entries in a Series are not NA.

```python
ser = pd.Series([5, 6, np.NaN])
```

```plaintext
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.0</td>
</tr>
<tr>
<td>1</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

```python
ser.notna()
```

```plaintext
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
</tr>
</tbody>
</table>
```

**pandas.Series.nsmallest**

Series.nsmallest(n=5, keep='first')

Return the smallest $n$ elements.

Parameters

- **n** [int, default 5] Return this many ascending sorted values.
- **keep** ['first', 'last', 'all'], default 'first' When there are duplicate values that cannot all fit in a Series of $n$ elements:
  - first: return the first $n$ occurrences in order of appearance.
  - last: return the last $n$ occurrences in reverse order of appearance.
  - all: keep all occurrences. This can result in a Series of size larger than $n$.

Returns
Series  The \( n \) smallest values in the Series, sorted in increasing order.

See also:

Series.nlargest  Get the \( n \) largest elements.
Series.sort_values  Sort Series by values.
Series.head  Return the first \( n \) rows.

Notes

Faster than .sort_values().head(n) for small \( n \) relative to the size of the Series object.

Examples

```python
>>> countries_population = {
  "Italy": 59000000,  "France": 65000000,
  "Brunei": 434000,    "Malta": 434000,
  "Maldives": 434000,  "Iceland": 337000,
  "Nauru": 11300,      "Tuvalu": 11300,
  "Anguilla": 11300,   "Montserrat": 5200
}
>>> s = pd.Series(countries_population)
>>> s
Italy     59000000
France    65000000
Brunei    434000
Malta     434000
Maldives  434000
Iceland   337000
Nauru     11300
Tuvalu    11300
Anguilla  11300
Montserrat 5200
dtype: int64

The \( n \) smallest elements where \( n=5 \) by default.

```python
>>> s.nsmallest()
Montserrat    5200
Nauru         11300
Tuvalu        11300
Anguilla      11300
Iceland       337000
dtype: int64
```

The \( n \) smallest elements where \( n=3 \). Default keep value is ‘first’ so Nauru and Tuvalu will be kept.

```python
>>> s.nsmallest(3)
Montserrat    5200
Nauru         11300
Tuvalu        11300
dtype: int64
```
The $n$ smallest elements where $n=3$ and keeping the last duplicates. Anguilla and Tuvalu will be kept since they are the last with value 11300 based on the index order.

```python
>>> s.nsmallest(3, keep='last')
Montserrat  5200
Anguilla  11300
Tuvalu  11300
dtype: int64
```

The $n$ smallest elements where $n=3$ with all duplicates kept. Note that the returned Series has four elements due to the three duplicates.

```python
>>> s.nsmallest(3, keep='all')
Montserrat  5200
Nauru  11300
Tuvalu  11300
Anguilla  11300
dtype: int64
```

**pandas.Series.nunique**

Series.nunique(dropna=True)

Return number of unique elements in the object.

Excludes NA values by default.

**Parameters**

- dropna [bool, default True] Don’t include NaN in the count.

**Returns**

- int

**See also:**

**DataFrame.nunique** Method nunique for DataFrame.

**Series.count** Count non-NA/null observations in the Series.

**Examples**

```python
>>> s = pd.Series([1, 3, 5, 7, 7])
>>> s
0  1
1  3
2  5
3  7
4  7
dtype: int64

>>> s.nunique()
4
```
pandas.Series.pad

Series.pad(axis=None, inplace=False, limit=None, downcast=None)

Synonym for DataFrame.fillna() with method='ffill'.

Returns

Series/DataFrame or None  Object with missing values filled or None if inplace=True.

pandas.Series.pct_change

Series.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods  [int, default 1] Periods to shift for forming percent change.
fill_method  [str, default 'pad'] How to handle NAs before computing percent changes.
limit  [int, default None] The number of consecutive NAs to fill before stopping.
freq  [DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).
**kwargs  Additional keyword arguments are passed into DataFrame.shift or Series.shift.

Returns

chg  [Series or DataFrame] The same type as the calling object.

See also:

Series.diff  Compute the difference of two elements in a Series.
DataFrame.diff  Compute the difference of two elements in a DataFrame.
Series.shift  Shift the index by some number of periods.
DataFrame.shift  Shift the index by some number of periods.

Examples

Series

```python
>>> s = pd.Series([90, 91, 85])
>>> s
0   90
1   91
2   85
dtype: int64
```
See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```python
>>> s = pd.Series([90, 91, None, 85])

>>> s
0    90.0
1    91.0
2    NaN
3    85.0
dtype: float64
```

```python
>>> s.pct_change(fill_method='ffill')
0    NaN
1    0.011111
2    0.000000
3    -0.065934
dtype: float64
```

**DataFrame**

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```python
>>> df = pd.DataFrame(
...     {'FR': [4.0405, 4.0963, 4.3149],
...      'GR': [1.7246, 1.7482, 1.8519],
...      'IT': [804.74, 810.01, 860.13],
...      'index': [1980-01-01, 1980-02-01, 1980-03-01])

>>> df
   FR     GR     IT
1980-01-01 4.0405 1.7246  804.74
1980-02-01 4.0963 1.7482  810.01
1980-03-01 4.3149 1.8519  860.13
```

```python
>>> df.pct_change()
   FR   GR   IT
1980-01-01  NaN  NaN  NaN
1980-02-01 0.013810 0.013684 0.006549
1980-03-01 0.053365 0.059318 0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

3.3. Series
```python
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351],
...     },
...     index=['GOOG', 'APPL'])
```

```python
>>> df
   2016  2015  2014
GOOG  1769950  1500923  1371819
APPL  30586265  40912316  41403351
```

```python
>>> df.pct_change(axis='columns', periods=-1)
   2016    2015    2014
GOOG   0.179241  0.094112   NaN
APPL  -0.252395 -0.011860   NaN
```

**pandas.Series.pipe**

`Series.pipe(func, *args, **kwargs)`

Apply chainable functions that expect Series or DataFrames.

**Parameters**

- `func` [function] Function to apply to the Series/DataFrame. `args` and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Series/DataFrame.

- `args` [iterable, optional] Positional arguments passed into `func`.

- `kwargs` [mapping, optional] A dictionary of keyword arguments passed into `func`.

**Returns**

- `object` [the return type of `func`].

**See also:**

- `DataFrame.apply` Apply a function along input axis of DataFrame.
- `DataFrame.applymap` Apply a function elementwise on a whole DataFrame.
- `Series.map` Apply a mapping correspondence on a `Series`.

**Notes**

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> func(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```python
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
```

(continues on next page)
If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```python
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((func, 'arg2'), arg1=a, arg3=c)
...)
```

### pandas.Series.plot

`Series.plot(*args, **kwargs)`

Make plots of Series or DataFrame.

Uses the backend specified by the option `plotting.backend`. By default, matplotlib is used.

**Parameters**

- **data** [Series or DataFrame] The object for which the method is called.
- **x** [label or position, default None] Only used if data is a DataFrame.
- **y** [label, position or list of label, positions, default None] Allows plotting of one column versus another. Only used if data is a DataFrame.
- **kind** [str] The kind of plot to produce:
  - ‘line’ : line plot (default)
  - ‘bar’ : vertical bar plot
  - ‘barh’ : horizontal bar plot
  - ‘hist’ : histogram
  - ‘box’ : boxplot
  - ‘kde’ : Kernel Density Estimation plot
  - ‘density’ : same as ‘kde’
  - ‘area’ : area plot
  - ‘pie’ : pie plot
  - ‘scatter’ : scatter plot (DataFrame only)
  - ‘hexbin’ : hexbin plot (DataFrame only)
- **ax** [matplotlib axes object, default None] An axes of the current figure.
- **subplots** [bool, default False] Make separate subplots for each column.
- **sharex** [bool, default True if `ax` is None else False] In case `subplots=True`, share x axis and set some x axis labels to invisible; defaults to True if `ax` is None otherwise False if an `ax` is passed in; Be aware, that passing in both an `ax` and `sharex=True` will alter all x axis labels for all axis in a figure.
- **sharey** [bool, default False] In case `subplots=True`, share y axis and set some y axis labels to invisible.
layout [tuple, optional] (rows, columns) for the layout of subplots.

figsize [a tuple (width, height) in inches] Size of a figure object.

use_index [bool, default True] Use index as ticks for x axis.

title [str or list] Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and subplots is True, print each item in the list above the corresponding subplot.

grid [bool, default None (matlab style default)] Axis grid lines.

legend [bool or ‘reverse’] Place legend on axis subplots.

style [list or dict] The matplotlib line style per column.

logx [bool or ‘sym’, default False] Use log scaling or symlog scaling on x axis. .. version-changed:: 0.25.0

logy [bool or ‘sym’ default False] Use log scaling or symlog scaling on y axis. .. version-changed:: 0.25.0

loglog [bool or ‘sym’, default False] Use log scaling or symlog scaling on both x and y axes. .. versionchanged:: 0.25.0

xticks [sequence] Values to use for the xticks.

yticks [sequence] Values to use for the yticks.

xlim [2-tuple/list] Set the x limits of the current axes.

ylim [2-tuple/list] Set the y limits of the current axes.

xlabel [label, optional] Name to use for the xlabel on x-axis. Default uses index name as xlabel, or the x-column name for planar plots.

New in version 1.1.0.

Changed in version 1.2.0: Now applicable to planar plots (scatter, hexbin).

ylabel [label, optional] Name to use for the ylabel on y-axis. Default will show no ylabel, or the y-column name for planar plots.

New in version 1.1.0.

Changed in version 1.2.0: Now applicable to planar plots (scatter, hexbin).

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots).

fontsize [int, default None] Font size for xticks and yticks.

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

colorbar [bool, optional] If True, plot colorbar (only relevant for ‘scatter’ and ‘hexbin’ plots).

position [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center).

table [bool, Series or DataFrame, default False] If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib’s default layout. If a Series or DataFrame is passed, use passed data to draw a table.

yerr [DataFrame, Series, array-like, dict and str] See Plotting with Error Bars for detail.

xerr [DataFrame, Series, array-like, dict and str] Equivalent to yerr.
stacked  [bool, default False in line and bar plots, and True in area plot] If True, create stacked plot.

sort_columns  [bool, default False] Sort column names to determine plot ordering.

secondary_y  [bool or sequence, default False] Whether to plot on the secondary y-axis if a list/tuple, which columns to plot on secondary y-axis.

mark_right  [bool, default True] When using a secondary_y axis, automatically mark the column labels with “(right)” in the legend.

include_bool  [bool, default is False] If True, boolean values can be plotted.

backend  [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend. New in version 1.0.0.

**kwargs  Options to pass to matplotlib plotting method.

Returns

matplotlib.axes.Axes or numpy.ndarray of them  If the backend is not the default matplotlib one, the return value will be the object returned by the backend.

Notes

• See matplotlib documentation online for more on this subject

• If kind = ‘bar’ or ‘barh’, you can specify relative alignments for bar plot layout by position keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

pandas.Series.pop

Series.pop(item)

Return item and drops from series. Raise KeyError if not found.

Parameters

item  [label] Index of the element that needs to be removed.

Returns

Value that is popped from series.

Examples

```python
>>> ser = pd.Series([1,2,3])

>>> ser.pop(0)
1

>>> ser
1  2
2  3
dtype: int64
```
pandas.Series.pow

Series.pow(other, level=None, fill_value=None, axis=0)

Return Exponential power of series and other, element-wise (binary operator pow).
Equivalent to series ** other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters
- other [Series or scalar value]
- fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns
- Series The result of the operation.

See also:
- Series.rpow Reverse of the Exponential power operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
```
```python
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
```
```python
>>> a.pow(b, fill_value=0)
a 1.0
b 1.0
c 1.0
d 0.0
e NaN
dtype: float64
```
pandas.Series.prod

Series.prod(axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs)

Return the product of the values over the requested axis.

Parameters

- **axis** [{index (0)}] Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **min_count** [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.
- **kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

- Series.sum Return the sum.
- Series.min Return the minimum.
- Series.max Return the maximum.
- Series.idxmin Return the index of the minimum.
- Series.idxmax Return the index of the maximum.
- DataFrame.sum Return the sum over the requested axis.
- DataFrame.min Return the minimum over the requested axis.
- DataFrame.max Return the maximum over the requested axis.
- DataFrame.idxmin Return the index of the minimum over the requested axis.
- DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the min_count parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the skipna parameter, min_count handles all-NA and empty series identically.
pandas.Series.product

`Series.product(axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs)`
Return the product of the values over the requested axis.

**Parameters**

- `axis` [{index (0)}] Axis for the function to be applied on.
- `skipna` [bool, default True] Exclude NA/null values when computing the result.
- `level` [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- `numeric_only` [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- `min_count` [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.
- `**kwargs` Additional keyword arguments to be passed to the function.

**Returns**

scalar or Series (if level specified)

**See also:**

- `Series.sum` Return the sum.
- `Series.min` Return the minimum.
- `Series.max` Return the maximum.
- `Series.idxmin` Return the index of the minimum.
- `Series.idxmax` Return the index of the maximum.
- `DataFrame.sum` Return the sum over the requested axis.
- `DataFrame.min` Return the minimum over the requested axis.
- `DataFrame.max` Return the maximum over the requested axis.
- `DataFrame.idxmin` Return the index of the minimum over the requested axis.
- `DataFrame.idxmax` Return the index of the maximum over the requested axis.
Examples

By default, the product of an empty or all-NA Series is 1

```python
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```python
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).prod()
1.0
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

**pandas.Series.quantile**

`Series.quantile(q=0.5, interpolation='linear')`

Return value at the given quantile.

**Parameters**

- `q` [float or array-like, default 0.5 (50% quantile)] The quantile(s) to compute, which can lie in range: 0 <= q <= 1.
- `interpolation` [‘linear’, ‘lower’, ‘higher’, ‘midpoint’, ‘nearest’] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points i and j:
  - linear: \( i + (j - i) \times \text{fraction} \), where \( \text{fraction} \) is the fractional part of the index surrounded by i and j.
  - lower: i.
  - higher: j.
  - nearest: i or j whichever is nearest.
  - midpoint: \((i + j) / 2\).

**Returns**

- float or Series If q is an array, a Series will be returned where the index is q and the values are the quantiles, otherwise a float will be returned.

**See also:**

- `core.window.Rolling.quantile` Calculate the rolling quantile.
- `numpy.percentile` Returns the q-th percentile(s) of the array elements.
Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25  1.75
0.50  2.50
0.75  3.25
dtype: float64
```

**pandas.Series.radd**

`Series.radd(other, level=None, fill_value=None, axis=0)`

Return Addition of series and other, element-wise (binary operator `radd`). Equivalent to `other + series`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- `other` [Series or scalar value]
- `fill_value` [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- `level` [int or name] Broadcast across a level, matching Index values on the passedMulti-Index level.

**Returns**

- `Series` The result of the operation.

**See also:**

- `Series.add` Element-wise Addition, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a   1.0
b   1.0
c   1.0
d   NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a   1.0
b   NaN
d   1.0
e   NaN
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
>>> a.add(b, fill_value=0)
a  2.0
b  1.0
c  1.0
d  1.0
e  NaN
dtype: float64
```

### pandas.Series.rank

`Series.rank(axis=0, method='average', numeric_only=False, na_option='keep', ascending=True, pct=False)`

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Index to direct ranking.
- **method** [‘average’, ‘min’, ‘max’, ‘first’, ‘dense’], default ‘average’] How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group
  - min: lowest rank in the group
  - max: highest rank in the group
  - first: ranks assigned in order they appear in the array
  - dense: like ‘min’, but rank always increases by 1 between groups.
- **numeric_only** [bool, optional] For DataFrame objects, rank only numeric columns if set to True.
- **na_option** [‘keep’, ‘top’, ‘bottom’], default ‘keep’] How to rank NaN values:
  - keep: assign NaN rank to NaN values
  - top: assign lowest rank to NaN values
  - bottom: assign highest rank to NaN values
- **ascending** [bool, default True] Whether or not the elements should be ranked in ascending order.
- **pct** [bool, default False] Whether or not to display the returned rankings in percentile form.

**Returns**

same type as caller Return a Series or DataFrame with data ranks as values.

**See also:**

- `core.groupby.GroupBy.rank` Rank of values within each group.
Examples

```python
>>> df = pd.DataFrame(data={
    'Animal': ['cat', 'penguin', 'dog', 'spider', 'snake'],
    'Number_legs': [4, 2, 4, 8, np.nan])

>>> df
    Animal  Number_legs
0      cat        4.0
1  penguin      2.0
2      dog        4.0
3    spider       8.0
4    snake       NaN
```

The following example shows how the method behaves with the above parameters:

- `default_rank`: this is the default behaviour obtained without using any parameter.
- `max_rank`: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since ‘cat’ and ‘dog’ are both in the 2nd and 3rd position, rank 3 is assigned.)
- `NA_bottom`: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- `pct_rank`: when setting `pct = True`, the ranking is expressed as percentile rank.

```python
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)

>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat        4.0          2.5        3.0        2.5    0.625
1  penguin      2.0          1.0        1.0        1.0    0.250
2      dog        4.0          2.5        3.0        2.5    0.625
3    spider       8.0          4.0        4.0        4.0    1.000
4    snake       NaN          NaN        NaN        NaN    NaN
```

**pandas.Series.ravel**

Series.ravel(order='C')

Return the flattened underlying data as an ndarray.

**Returns**

- `numpy.ndarray` or `ndarray-like` Flattened data of the Series.

**See also:**

- `numpy.ndarray.ravel` Return a flattened array.
pandas.Series.rdiv

**pandas.Series.rdiv(other, level=None, fill_value=None, axis=0)**

Return Floating division of series and other, element-wise (binary operator `rtruediv`).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- **Series** The result of the operation.

**See also:**

- **Series.truediv** Element-wise Floating division, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64

>>> a.divide(b, fill_value=0)
    a    b
a  1.0   inf
b   inf   inf
c   inf   inf
d  0.0  NaN
e  NaN   NaN
dtype: float64
```
**pandas.Series.rdivmod**

Series.rdivmod(other, level=None, fill_value=None, axis=0)

Return Integer division and modulo of series and other, element-wise (binary operator rdivmod).

Equivalent to other % series, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

- other [Series or scalar value]
- fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

2-Tuple of Series The result of the operation.

**See also:**

Series.divmod Element-wise Integer division and modulo, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0  
b  1.0  
c  1.0  
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0  
b  NaN  
d  1.0  
e  NaN
dtype: float64
>>> a.divmod(b, fill_value=0)
(a  1.0  
b  NaN  
c  NaN  
d  0.0  
e  NaN
dtype: float64,
(a  0.0  
b  NaN  
c  NaN  
d  0.0  
(continues on next page)
```
pandas.Series.reindex

Series.reindex(*args, **kwargs)
Conform Series to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

index [array-like, optional] New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data.

method [[None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’]] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
• None (default): don’t fill gaps
• pad / ffill: Propagate last valid observation forward to next valid.
• backfill / bfill: Use next valid observation to fill gap.
• nearest: Use nearest valid observations to fill gap.

copy [bool, default True] Return a new object, even if the passed indexes are the same.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any “compatible” value.

limit [int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation \( \text{abs(index[indexer] - target)} \leq \text{tolerance} \).
Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

Returns

Series with changed index.

See also:

DataFrame.set_index Set row labels.
DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.
**Examples**

DataFrame.reindex supports two calling conventions

- (index=index_labels, columns=column_labels, ...)
- (labels, axis={‘index’, ‘columns’}, ...)

We highly recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```python
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
```

```
<table>
<thead>
<tr>
<th>http_status</th>
<th>response_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>200</td>
</tr>
<tr>
<td>Chrome</td>
<td>200</td>
</tr>
<tr>
<td>Safari</td>
<td>404</td>
</tr>
<tr>
<td>IE10</td>
<td>404</td>
</tr>
<tr>
<td>Konqueror</td>
<td>301</td>
</tr>
</tbody>
</table>
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

```python
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10', 'Chrome']
>>> df.reindex(new_index)
```

```
<table>
<thead>
<tr>
<th>http_status</th>
<th>response_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>404.0</td>
</tr>
<tr>
<td>Iceweasel</td>
<td>NaN</td>
</tr>
<tr>
<td>Comodo Dragon</td>
<td>NaN</td>
</tr>
<tr>
<td>IE10</td>
<td>404.0</td>
</tr>
<tr>
<td>Chrome</td>
<td>200.0</td>
</tr>
</tbody>
</table>
```

We can fill in the missing values by passing a value to the keyword fill_value. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword method to fill the NaN values.

```python
>>> df.reindex(new_index, fill_value=0)
```

```
<table>
<thead>
<tr>
<th>http_status</th>
<th>response_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>404</td>
</tr>
<tr>
<td>Iceweasel</td>
<td>0</td>
</tr>
<tr>
<td>Comodo Dragon</td>
<td>0</td>
</tr>
<tr>
<td>IE10</td>
<td>404</td>
</tr>
<tr>
<td>Chrome</td>
<td>200</td>
</tr>
</tbody>
</table>
```

```python
>>> df.reindex(new_index, fill_value='missing')
```

```
<table>
<thead>
<tr>
<th>http_status</th>
<th>response_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>404</td>
</tr>
<tr>
<td>Iceweasel</td>
<td>missing</td>
</tr>
<tr>
<td>Comodo Dragon</td>
<td>missing</td>
</tr>
<tr>
<td>IE10</td>
<td>404</td>
</tr>
<tr>
<td>Chrome</td>
<td>200</td>
</tr>
</tbody>
</table>
```
We can also reindex the columns.

```python
>>> df.reindex(columns=['http_status', 'user_agent'])
                     http_status  user_agent
Firefox           200       NaN
Chrome            200       NaN
Safari            404       NaN
IE10              404       NaN
Konqueror         301       NaN
```

Or we can use “axis-style” keyword arguments

```python
>>> df.reindex([['http_status', 'user_agent'], axis="columns")
                     http_status  user_agent
Firefox           200       NaN
Chrome            200       NaN
Safari            404       NaN
IE10              404       NaN
Konqueror         301       NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```python
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({'prices': [100, 101, np.nan, 100, 89, 88]}, index=date_index)
>>> df2
   prices
2010-01-01  100.0
2010-01-02  101.0
2010-01-03  NaN
2010-01-04  100.0
2010-01-05  89.0
2010-01-06  88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```python
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
   prices
2009-12-29  NaN
2009-12-30  NaN
2009-12-31  NaN
2010-01-01  100.0
2010-01-02  101.0
2010-01-03  NaN
2010-01-04  100.0
2010-01-05  89.0
2010-01-06  88.0
2010-01-07  NaN
```

The index entries that did not have a value in the original data frame (for example, ‘2009-12-29’) are by default filled with NaN. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the NaN values, pass `bfill` as an argument to
the method keyword.

```python
>>> df2.reindex(date_index2, method='bfill')
prices
2009-12-29  100.0
2009-12-30  100.0
2009-12-31  100.0
2010-01-01  100.0
2010-01-02  101.0
2010-01-03  NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
2010-01-07  NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

**pandas.Series.reindex_like**

`Series.reindex_like(other, method=None, copy=True, limit=None, tolerance=None)`

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

**Parameters**

- **other** [Object of the same data type] Its row and column indices are used to define the new indices of this object.
- **method** [None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
  - None (default): don’t fill gaps
  - pad / ffill: propagate last valid observation forward to next valid
  - backfill / bfill: use next valid observation to fill gap
  - nearest: use nearest valid observations to fill gap.
- **copy** [bool, default True] Return a new object, even if the passed indexes are the same.
- **limit** [int, default None] Maximum number of consecutive labels to fill for inexact matches.
- **tolerance** [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple,
array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

Returns

Series or DataFrame  Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index  Set row labels.
DataFrame.reset_index  Remove row labels or move them to new columns.
DataFrame.reindex  Change to new indices or expand indices.

Notes

Same as calling .reindex(index=other.index, columns=other.columns,...).

Examples

```python
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
                      [31, 87.8, 'high'],
                      [22, 71.6, 'medium'],
                      [35, 95, 'medium']],
                     columns=['temp_celsius', 'temp_fahrenheit', 'windspeed'],
                     index=pd.date_range(start='2014-02-12',
                                          end='2014-02-15', freq='D'))
```

```python
>>> df2 = pd.DataFrame([[28, 'low'],
                      [30, 'low'],
                      [35.1, 'medium']],
                     columns=['temp_celsius', 'windspeed'],
                     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13', '2014-02-15']))
```

```python
>>> df2.reindex_like(df1)
```

(continues on next page)
pandas.Series.rename

Series.rename(index=None, *, axis=None, copy=True, inplace=False, level=None, errors='ignore')

Alter Series index labels or name.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don’t throw an error.

Alternatively, change Series.name with a scalar value.

See the user guide for more.

Parameters

- axis [{0 or “index”}] Unused. Accepted for compatibility with DataFrame method only.
- index [scalar, hashable sequence, dict-like or function, optional] Functions or dict-like are transformations to apply to the index. Scalar or hashable sequence-like will alter the Series.name attribute.
- **kwargs Additional keyword arguments passed to the function. Only the “inplace” keyword is used.

Returns

Series or None Series with index labels or name altered or None if inplace=True.

See also:

DataFrame.rename Corresponding DataFrame method.

Series.rename_axis Set the name of the axis.

Examples

```python
>>> s = pd.Series([1, 2, 3])
>>> s
0  1
1  2
2  3
dtype: int64
>>> s.rename("my_name")  # scalar, changes Series.name
0  1
1  2
2  3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0  1
1  4
3
dtype: int64
```
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0  1
3  2
5  3
dtype: int64

**pandas.Series.rename_axis**

Series.rename_axis(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False)

Set the name of the axis for the index or columns.

**Parameters**

mapper [scalar, list-like, optional] Value to set the axis name attribute.

index, columns [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis’ values. Note that the columns parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either mapper and axis to specify the axis to target with mapper, or index and/or columns.

axis [(0 or 'index', 1 or 'columns'), default 0] The axis to rename.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

**Returns**

Series, DataFrame, or None The same type as the caller or None if inplace=True.

**See also:**

Series.rename Alter Series index labels or name.

DataFrame.rename Alter DataFrame index labels or name.

Index.rename Set new names on index.

**Notes**

DataFrame.rename_axis supports two calling conventions

* (index=index_mapper, columns=columns_mapper, ...)
* (mapper, axis={'index', 'columns'}, ...)

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter copy is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis labels.

We highly recommend using keyword arguments to clarify your intent.
Examples

Series

```python
>>> s = pd.Series(['dog', 'cat', 'monkey'])
>>> s
0    dog
1     cat
2   monkey
dtype: object

>>> s.rename_axis("animal")
animal
0    dog
1     cat
2   monkey
dtype: object
```

Dataframe

```python
>>> df = pd.DataFrame({'num_legs': [4, 4, 2],
                     'num_arms': [0, 0, 2]},
                    index=['dog', 'cat', 'monkey'])
>>> df
    num_legs  num_arms
dog       4       0
cat       4       0
monkey    2       2
>>> df = df.rename_axis("animal")
```

```python
>>> df
animal
    num_legs  num_arms
dog       4       0
cat       4       0
monkey    2       2
```

```python
>>> df = df.rename_axis("limbs", axis="columns")
```

```python
>>> df
    num_legs  num_arms
animal
  limbs  num_legs  num_arms
    dog       4       0
    cat       4       0
  monkey    2       2
```

MultiIndex

```python
>>> df.index = pd.MultiIndex.from_product([["mammal"],
                                         ['dog', 'cat', 'monkey'],
                                         names=['type', 'name'])
```

```python
>>> df
animal
    num_legs  num_arms
  limbs  type  name  num_legs  num_arms
  mammal  dog      4       0
          cat      4       0
          monkey  2       2
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.rename_axis(index={'type': 'class'})
limbs  num_legs  num_arms
class name
mammal dog   4   0
    cat       4   0
  monkey    2   2

>>> df.rename_axis(columns=str.upper)
LIMBS  num_legs  num_arms
type  name
mammal dog   4   0
    cat       4   0
  monkey    2   2
```

**pandas.Series.reorder_levels**

Series.reorder_levels(order)
Rearrange index levels using input order.
May not drop or duplicate levels.

- **Parameters**
  - order [list of int representing new level order] Reference level by number or key.

- **Returns**
  - type of caller (new object)

**pandas.Series.repeat**

Series.repeat(repeats, axis=None)
Repeat elements of a Series.

Returns a new Series where each element of the current Series is repeated consecutively a given number of times.

- **Parameters**
  - repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Series.
  - axis [None] Must be None. Has no effect but is accepted for compatibility with numpy.

- **Returns**
  - Series Newly created Series with repeated elements.

**See also:**

- Index.repeat Equivalent function for Index.
- numpy.repeat Similar method for numpy.ndarray.
Examples

```python
>>> s = pd.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
dtype: object
>>> s.repeat(2)
0    a
1    a
1    b
1    b
2    c
2    c
dtype: object
>>> s.repeat([1, 2, 3])
0    a
1    b
1    b
2    c
2    c
2    c
dtype: object
```

**pandas.Series.replace**

Series.replace(to_replace=None, value=NoDefault.no_default, inplace=False, limit=None, regex=False, method=NoDefault.no_default)

Replace values given in `to_replace` with `value`.

Values of the Series are replaced with other values dynamically.

This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

**Parameters**

- **to_replace** [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.
  - numeric, str or regex:
    - numeric: numeric values equal to `to_replace` will be replaced with `value`
    - str: string exactly matching `to_replace` will be replaced with `value`
    - regex: regexs matching `to_replace` will be replaced with `value`
  - list of str, regex, or numeric:
    - First, if `to_replace` and `value` are both lists, they must be the same length.
    - Second, if `regex=True` then all of the strings in both lists will be interpreted as regexes otherwise they will match directly. This doesn’t matter much for `value` since there are only a few possible substitution regexes you can use.
    - str, regex and numeric rules apply as above.
• dict:
  – Dicts can be used to specify different replacement values for different existing
    values. For example, `{‘a’: ‘b’, ‘y’: ‘z’}` replaces the value ‘a’ with ‘b’
    and ‘y’ with ‘z’. To use a dict in this way the `value` parameter should be `None`.
  – For a DataFrame a dict can specify that different values should be replaced in
    different columns. For example, `{‘a’: 1, ‘b’: ‘z’}` looks for the value 1 in
    column ‘a’ and the value ‘z’ in column ‘b’ and replaces these values with whatever
    is specified in `value`. The `value` parameter should not be `None` in this case.
    You can treat this as a special case of passing two lists except that you are specifying
    the column to search in.
  – For a DataFrame nested dictionaries, e.g., `{‘a’: {‘b’: np.nan}}`, are read as
    follows: look in column ‘a’ for the value ‘b’ and replace it with NaN. The `value`
    parameter should be `None` to use a nested dict in this way. You can nest regular
    expressions as well. Note that column names (the top-level dictionary keys in a
    nested dictionary) **cannot** be regular expressions.

• `None`:
  – This means that the `regex` argument must be a string, compiled regular expression,
    or list, dict, ndarray or Series of such elements. If `value` is also `None` then this
    must be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** [scalar, dict, list, str, regex, default None] Value to replace any values matching
`to_replace` with. For a DataFrame a dict of values can be used to specify which value
the `value` parameter should be `None` to use a nested dict in this way. You can nest regular
expressions as well. Note that column names (the top-level dictionary keys in a
nested dictionary) **cannot** be regular expressions.

**inplace** [bool, default False] If True, performs operation inplace and returns None.

**limit** [int, default None] Maximum size gap to forward or backward fill.

**regex** [bool or same types as `to_replace`, default False] Whether to interpret `to_replace`
and/or `value` as regular expressions. If this is `True` then `to_replace` must be a string.
Alternatively, this could be a regular expression or a list, dict, or array of regular ex-
pressions in which case `to_replace` must be `None`.

**method** [{‘pad’, ‘ffill’, ‘bfill’, None}] The method to use when for replacement, when
`to_replace` is a scalar, list or tuple and `value` is `None`.

Changed in version 0.23.0: Added to DataFrame.

**Returns**

**Series** Object after replacement.

**Raises**

**AssertionError**
  • If `regex` is not a bool and `to_replace` is not `None`.

**TypeError**
  • If `to_replace` is not a scalar, array-like, dict, or `None`
  • If `to_replace` is a dict and `value` is not a list, dict, ndarray, or Series
  • If `to_replace` is `None` and `regex` is not compilable into a regular expression or is a
    list, dict, ndarray, or Series.
When replacing multiple bool or datetime64 objects and the arguments to `to_replace` does not match the type of the value being replaced

ValueError

- If a list or an ndarray is passed to `to_replace` and `value` but they are not the same length.

See also:

- `Series.fillna` Fill NA values.
- `Series.where` Replace values based on boolean condition.
- `Series.str.replace` Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers are strings, then you can do this.
- This method has a lot of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the `to_replace` value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar `to_replace` and `value`

```python
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```python
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
                      'B': [5, 6, 7, 8, 9],
                      'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
    A  B  C
0    5  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```
List-like `to_replace`

```python
>>> df.replace([0, 1, 2, 3], 4)
     A  B  C
0   4  5  a
1   4  6  b
2   4  7  c
3   4  8  d
4   4  9  e
```

```python
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
     A  B  C
0   4  5  a
1   3  6  b
2   2  7  c
3   1  8  d
4   4  9  e
```

```python
>>> s.replace([1, 2], method='bfill')
0    3
1    3
2    3
3    4
4    5
dtype: int64
```

dict-like `to_replace`

```python
>>> df.replace({0: 10, 1: 100})
     A  B  C
0  10  5  a
1 100  6  b
2   2  7  c
3   3  8  d
4   4  9  e
```

```python
>>> df.replace({'A': 0, 'B': 5}, 100)
     A  B  C
0  100 100  a
1    1  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```python
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression `to_replace`

```python
>>> df.replace({'A': r'\ Al', 'B': r'\ B1'}, 100)
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
   A    B
0  new abc
1    foo new
2   bait xyz

>>> df.replace({'A': r'^ba.$', 'A': 'new'}, regex=True)
   A    B
0  new abc
1    foo bar
2   bait xyz

>>> df.replace(regex=r'^ba.$', value='new')
   A    B
0  new abc
1    foo new
2   bait xyz

>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
   A    B
0  new abc
1    xyz new
2   bait xyz

>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
   A    B
0  new abc
1      new
2   bait xyz

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```python
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the `value` parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None):

```python
>>> s.replace({'a': None})
0   10
1   None
2   None
3    b
4   None
dtype: object
```

When `value` is not explicitly passed and `to_replace` is a scalar, list or tuple, `replace` uses the method parameter (default ‘pad’) to do the replacement. So this is why the ‘a’ values are being replaced by 10 in rows 1 and 2 and ‘b’ in row 4 in this case.
On the other hand, if `None` is explicitly passed for `value`, it will be respected:

```python
>>> s.replace('a', None)
0     10
1   None
2   None
3     b
4   None
dtype: object
```

Changed in version 1.4.0: Previously the explicit `None` was silently ignored.

**pandas.Series.resample**

`Series.resample`(rule, axis=0, closed=None, label=None, convention='start', kind=None, loffset=None, base=None, on=None, level=None, origin='start_day', offset=None)

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`, or `TimedeltaIndex`), or the caller must pass the label of a datetime-like series/index to the `on/level` keyword parameter.

**Parameters**

- **rule** [DateOffset, Timedelta or str] The offset string or object representing target conversion.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Which axis to use for up- or downsampling. For Series this will default to 0, i.e. along the rows. Must be `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
- **closed** [‘right’, ‘left’], default None Which side of bin interval is closed. The default is ‘left’ for all frequency offsets except for ‘M’, ‘A’, ‘Q’, ‘BM’, ‘BA’, ‘BQ’, and ‘W’ which all have a default of ‘right’.
- **label** [‘right’, ‘left’], default None Which bin edge label to label bucket with. The default is ‘left’ for all frequency offsets except for ‘M’, ‘A’, ‘Q’, ‘BM’, ‘BA’, ‘BQ’, and ‘W’ which all have a default of ‘right’.
- **convention** [‘start’, ‘end’, ‘s’, ‘e’], default ‘start’ For `PeriodIndex` only, controls whether to use the start or end of `rule`.
- **kind** [‘timestamp’, ‘period’], optional, default None Pass ‘timestamp’ to convert the resulting index to a `DateTimeIndex` or ‘period’ to convert it to a `PeriodIndex`. By default the input representation is retained.
- **loffset** [timedelta, default None] Adjust the resampled time labels.
Deprecated since version 1.1.0: You should add the loffset to the df.index after the resample. See below.

**base** [int, default 0] For frequencies that evenly subdivide 1 day, the “origin” of the aggregated intervals. For example, for ‘5min’ frequency, base could range from 0 through 4. Defaults to 0.

Deprecated since version 1.1.0: The new arguments that you should use are ‘offset’ or ‘origin’.

**on** [str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

**level** [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. level must be datetime-like.

**origin** [Timestamp or str, default ‘start_day’] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:

- ‘epoch’: origin is 1970-01-01
- ‘start’: origin is the first value of the timeseries
- ‘start_day’: origin is the first day at midnight of the timeseries

New in version 1.1.0.

- ‘end’: origin is the last value of the timeseries
- ‘end_day’: origin is the ceiling midnight of the last day

New in version 1.3.0.

**offset** [Timedelta or str, default is None] An offset timedelta added to the origin.

New in version 1.1.0.

### Returns

pandas.core.Resampler  Resampler object.

See also:

- **Series.resample**  Resample a Series.
- **DataFrame.resample**  Resample a DataFrame.
- **groupby**  Group Series by mapping, function, label, or list of labels.
- **asfreq**  Reindex a Series with the given frequency without grouping.

### Notes

See the user guide for more.

To learn more about the offset strings, please see this link.
Examples

Start by creating a series with 9 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```python
>>> series.resample('3T').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```python
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00    3
2000-01-01 00:06:00   12
2000-01-01 00:09:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```python
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```python
>>> series.resample('30S').asfreq()[0:5]  # Select first 5 rows
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    1.0
```

(continues on next page)
Upsample the series into 30 second bins and fill the NaN values using the pad method.

```python
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00 0
2000-01-01 00:00:30 0
2000-01-01 00:01:00 1
2000-01-01 00:01:30 1
2000-01-01 00:02:00 2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the NaN values using the bfill method.

```python
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00 0
2000-01-01 00:00:30 1
2000-01-01 00:01:00 1
2000-01-01 00:01:30 2
2000-01-01 00:02:00 2
Freq: 30S, dtype: int64
```

Pass a custom function via apply

```python
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
...     >>>

>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00 8
2000-01-01 00:03:00 17
2000-01-01 00:06:00 26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword `convention` can be used to control whether to use the start or end of rule.

Resample a year by quarter using `start` convention. Values are assigned to the first quarter of the period.

```python
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...     freq='A',
...     periods=2))

>>> s
2012  1
2013  2
Freq: A-DEC, dtype: int64

>>> s.resample('Q', convention='start').asfreq()
2012Q1  1.0
2012Q2  NaN
2012Q3  NaN
2012Q4  NaN
2013Q1  2.0
```

(continues on next page)
Resample quarters by month using ‘end’ convention. Values are assigned to the last month of the period.

```python
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',... freq='Q',... periods=4))
>>> q
2018Q1 1
2018Q2 2
2018Q3 3
2018Q4 4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03 1.0
2018-04 NaN
2018-05 NaN
2018-06 2.0
2018-07 NaN
2018-08 NaN
2018-09 3.0
2018-10 NaN
2018-11 NaN
2018-12 4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword on can be used to specify the column instead of the index for resampling.

```python
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],...
'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',...
periods=8,  
freq='W')
>>> df
   price  volume  week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2     9       40    2018-01-21
3    13      100    2018-01-28
4    14       50    2018-02-04
5     18      100    2018-02-11
6    17       40    2018-02-18
7     19       50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
   price  volume
week_starting
2018-01-31  10.75   62.5
2018-02-28  17.00   60.0
```
For a DataFrame with MultiIndex, the keyword level can be used to specify on which level the resampling needs to take place.

```python
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 17, 19],
...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(
...   d2,
...   index=pd.MultiIndex.from_product(
...     [days, ['morning', 'afternoon']]
...   )
... )
```

```python
>>> df2
   price  volume
2000-01-01 morning    10     50
         afternoon    11     60
2000-01-02 morning    9     40
         afternoon   13    100
2000-01-03 morning   14     50
         afternoon   18    100
2000-01-04 morning   17     40
         afternoon   19     50
```

```python
>>> df2.resample('D', level=0).sum()
   price  volume
2000-01-01    21    110
2000-01-02    22    140
2000-01-03    32    150
2000-01-04    36    90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```python
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00    0
2000-10-01 23:37:00    3
2000-10-01 23:44:00    6
2000-10-01 23:51:00    9
2000-10-01 23:58:00   12
2000-10-02 00:05:00   15
2000-10-02 00:12:00   18
2000-10-02 00:19:00   21
2000-10-02 00:26:00   24
Freq: 7T, dtype: int64
```

```python
>>> ts.resample('17min').sum()
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
2000-10-02 00:22:00   24
Freq: 17T, dtype: int64
```
If you want to adjust the start of the bins with an offset Timedelta, the two following lines are equivalent:

```python
>>> ts.resample('17min', origin='start').sum()
```
```
2000-10-01 23:30:00   9
2000-10-01 23:47:00   21
2000-10-02 00:04:00   54
2000-10-02 00:21:00   24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```python
>>> ts.resample('17min', origin='end').sum()
```
```
2000-10-01 23:38:00   3
2000-10-01 23:55:00   15
2000-10-02 00:12:00   45
2000-10-02 00:29:00   45
Freq: 17T, dtype: int64
```

In contrast with the `start_day`, you can use `end_day` to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```python
>>> ts.resample('17min', origin='end_day').sum()
```
```
2000-10-01 23:35:00   0
2000-10-01 23:52:00   18
2000-10-02 00:09:00   27
2000-10-02 00:26:00   63
Freq: 17T, dtype: int64
```

To replace the use of the deprecated `base` argument, you can now use `offset`, in this example it is equivalent to have `base=2`:


```python
>>> ts.resample('17min', offset='2min').sum()
2000-10-01 23:16:00    0
2000-10-01 23:33:00    9
2000-10-01 23:50:00   36
2000-10-02 00:07:00   39
2000-10-02 00:24:00   24
Freq: 17T, dtype: int64
```

To replace the use of the deprecated `loffset` argument:

```python
>>> from pandas.tseries.frequencies import to_offset
>>> loffset = '19min'
>>> ts_out = ts.resample('17min').sum()
>>> ts_out.index = ts_out.index + to_offset(loffset)
>>> ts_out
2000-10-01 23:33:00    0
2000-10-01 23:50:00    9
2000-10-02 00:07:00   21
2000-10-02 00:24:00   54
2000-10-02 00:41:00   24
Freq: 17T, dtype: int64
```

**pandas.Series.reset_index**

`Series.reset_index(level=None, drop=False, name=None, inplace=False)`

Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

**Parameters**

- **level** [int, str, tuple, or list, default optional] For a Series with a MultiIndex, only remove the specified levels from the index. Removes all levels by default.
- **drop** [bool, default False] Just reset the index, without inserting it as a column in the new DataFrame.
- **name** [object, optional] The name to use for the column containing the original Series values. Uses `self.name` by default. This argument is ignored when `drop` is True.
- **inplace** [bool, default False] Modify the Series in place (do not create a new object).

**Returns**

- **Series or DataFrame or None** When `drop` is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When `drop` is True, a `Series` is returned. In either case, if `inplace=True`, no value is returned.

**See also:**

- `DataFrame.reset_index` Analogous function for DataFrame.
Examples

```python
>>> s = pd.Series([1, 2, 3, 4], name='foo',
                 index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```python
>>> s.reset_index()
    idx  foo
 0   a    1
 1   b    2
 2   c    3
 3   d    4
```

To specify the name of the new column use `name`.

```python
>>> s.reset_index(name='values')
    idx  values
 0   a    1
 1   b    2
 2   c    3
 3   d    4
```

To generate a new Series with the default set `drop` to True.

```python
>>> s.reset_index(drop=True)
 0 1
 1 2
 2 3
 3 4
Name: foo, dtype: int64
```

To update the Series in place, without generating a new one set `inplace` to True. Note that it also requires `drop=True`.

```python
>>> s.reset_index(inplace=True, drop=True)
>>> s
 0 1
 1 2
 2 3
 3 4
Name: foo, dtype: int64
```

The `level` parameter is interesting for Series with a multi-level index.

```python
>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
            np.array(['one', 'two', 'one', 'two'])]
>>> s2 = pd.Series(...
                 range(4), name='foo',
                 index=pd.MultiIndex.from_arrays(arrays,
                 names=['a', 'b'])))
```

To remove a specific level from the Index, use `level`.
>>> s2.reset_index(level='a')
  a  foo
  b
  one  bar  0
  two  bar  1
  one  baz  2
  two  baz  3

If level is not set, all levels are removed from the Index.

>>> s2.reset_index()
  a  b  foo
  0  bar one  0
  1  bar two  1
  2  baz one  2
  3  baz two  3

**pandas.Series.rfloordiv**

*pandas.Series.rfloordiv*(other, level=None, fill_value=None, axis=0)

Return Integer division of series and other, element-wise (binary operator rfloordiv).

Equivalent to other // series, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

Series The result of the operation.

**See also:**

*pandas.Series.floordiv* Element-wise Integer division, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
```
```python
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
```
pandas.Series.rmul

Series.rmul(other, level=None, fill_value=None, axis=0)

Return Multiplication of series and other, element-wise (binary operator rmul).

Equivalent to other * series, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

other [Series or scalar value]
fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns

Series The result of the operation.

See also:

Series.mul Element-wise Multiplication, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
```

```python
>>> a.multiply(b, fill_value=0)
a  1.0
b  0.0
c  0.0
d  0.0
e  NaN
dtype: float64
```

### pandas.Series.rolling

**Series.rolling**(window, min_periods=None, center=False, win_type=None, on=None, axis=0, closed=None, method='single')

Provide rolling window calculations.

**Parameters**

- **window** [int, offset, or BaseIndexer subclass] Size of the moving window.
  - If an integer, the fixed number of observations used for each window.
  - If an offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. To learn more about the offsets & frequency strings, please see this link.
  - If a BaseIndexer subclass, the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely `min_periods`, `center`, and `closed` will be passed to `get_window_bounds`.

- **min_periods** [int, default None] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.
  - For a window that is specified by an offset, `min_periods` will default to 1.
  - For a window that is specified by an integer, `min_periods` will default to the size of the window.

- **center** [bool, default False] If False, set the window labels as the right edge of the window index.
  - If True, set the window labels as the center of the window index.

- **win_type** [str, default None] If None, all points are evenly weighted.
  - If a string, it must be a valid `scipy.signal` window function.
  - Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature.

- **on** [str, optional] For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame’s index.

3.3. Series 1367
Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

**axis** [int or str, default 0] If 0 or 'index', roll across the rows.
   If 1 or 'columns', roll across the columns.

**closed** [str, default None] If 'right', the first point in the window is excluded from calculations.
   If 'left', the last point in the window is excluded from calculations.
   If 'both', the no points in the window are excluded from calculations.
   If 'neither', the first and last points in the window are excluded from calculations.
   Default None ('right').
   Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

**method** [str {'single', 'table'}, default 'single'] New in version 1.3.0.
   Execute the rolling operation per single column or row ('single') or over the entire object ('table').
   This argument is only implemented when specifying engine='numba' in the method call.

**Returns**

- **Window subclass if a win_type is passed**
- **Rolling subclass if win_type is not passed**

**See also:**

- `expanding` Provides expanding transformations.
- `ewm` Provides exponential weighted functions.

**Notes**

See *Windowing Operations* for further usage details and examples.

**Examples**

```python
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```python
>>> window
Rolling sum with a window length of 2 observations.
```
>>> df.rolling(2).sum()
   B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN

Rolling sum with a window span of 2 seconds.

```python
>>> df_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
                        index=[pd.Timestamp('20130101 09:00:00'),
                               pd.Timestamp('20130101 09:00:02'),
                               pd.Timestamp('20130101 09:00:03'),
                               pd.Timestamp('20130101 09:00:05'),
                               pd.Timestamp('20130101 09:00:06')])
```

```python
>>> df_time
      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

```python
>>> df_time.rolling('2s').sum()
      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Rolling sum with forward looking windows with 2 observations.

```python
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
   B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```

min_periods

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```python
>>> df.rolling(2, min_periods=1).sum()
   B
0  0.0
1  1.0
```

(continues on next page)
center
Rolling sum with the result assigned to the center of the window index.

```python
>>> df.rolling(3, min_periods=1, center=True).sum()
   B
0   1.0
1   3.0
2   3.0
3   6.0
4   4.0
```

```python
>>> df.rolling(3, min_periods=1, center=False).sum()
   B
0   0.0
1   1.0
2   3.0
3   3.0
4   6.0
```

win_type
Rolling sum with a window length of 2, using the Scipy 'gaussian' window type. std is required in the aggregation function.

```python
>>> df.rolling(2, win_type='gaussian').sum(std=3)
   B
0   NaN
1   0.986207
2   2.958621
3   NaN
4   NaN
```

pandas.Series.round

Series.round(decimals=0, *args, **kwargs)
Round each value in a Series to the given number of decimals.

Parameters
- `decimals` [int, default 0] Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point.

- *args, **kwargs Additional arguments and keywords have no effect but might be accepted for compatibility with NumPy.

Returns
- `Series` Rounded values of the Series.

See also:
**numpy.around**  Round values of an np.array.

**DataFrame.round**  Round values of a DataFrame.

**Examples**

```python
>>> s = pd.Series([0.1, 1.3, 2.7])
>>> s.round()
 0   0.0
 1   1.0
 2   3.0
dtype: float64
```

**pandas.Series.rpow**

**Series.rpow**(other, level=None, fill_value=None, axis=0)

Return Exponential power of series and other, element-wise (binary operator rpow).

Equivalent to other ** series, but with support to substitute a fill_value for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **Series** The result of the operation.

**See also:**

**Series.pow**  Element-wise Exponential power, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
  a   1.0
  b   1.0
  c   1.0
  d   NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
  a   1.0
  b   NaN
```
pandas.Series.rsub

Series.rsub\(\text{other, level=None, fill_value=None, axis=0}\)  
Return Subtraction of series and other, element-wise (binary operator rsub).  
Equivalent to \text{other} - \text{series}, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

- \text{other} [Series or scalar value]
- \text{fill_value} [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- \text{level} [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns

Series The result of the operation.

See also:

Series.sub Element-wise Subtraction, see Python documentation for more details.

Examples

```py
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```python
e NaN
dtype: float64

a  0.0
b  1.0
c  1.0
d  -1.0
e  NaN
dtype: float64
```

### pandas.Series.rtruediv

**Series.rtruediv(other, level=None, fill_value=None, axis=0)**

Return Floating division of series and other, element-wise (binary operator `rtruediv`).

Equivalent to `other / series`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- **Series** The result of the operation.

**See also:**

- **Series.truediv** Element-wise Floating division, see Python documentation for more details.

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0
e  NaN
```

(continues on next page)
pandas.Series.sample

Series.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None, ignore_index=False)

Return a random sample of items from an axis of object.

You can use random_state for reproducibility.

Parameters

- n [int, optional] Number of items from axis to return. Cannot be used with frac. Default = 1 if frac = None.
- frac [float, optional] Fraction of axis items to return. Cannot be used with n.
- replace [bool, default False] Allow or disallow sampling of the same row more than once.
- weights [str or ndarray-like, optional] Default ‘None’ results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.
- random_state [int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

Changed in version 1.1.0: array-like and BitGenerator object now passed to np.random.RandomState() as seed

Changed in version 1.4.0: np.random.Generator objects now accepted
- axis [[0 or ‘index’, 1 or ‘columns’, None], default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames).
- ignore_index [bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

New in version 1.3.0.

Returns

Series or DataFrame A new object of same type as caller containing n items randomly sampled from the caller object.
See also:

DataFrameGroupBy.sample  Generates random samples from each group of a DataFrame object.
SeriesGroupBy.sample  Generates random samples from each group of a Series object.
numpy.random.choice  Generates a random sample from a given 1-D numpy array.

Notes

If frac > 1, replacement should be set to True.

Examples

```python
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>>
>>> df
   num_legs  num_wings  num_specimen_seen
falcon     2          2               10
  dog       4          0                2
spider     8          0                1
falcon     2          2               10
falcon     2          2               10
  fish      0          0                8
  dog       4          0                2
falcon     2          2               10
falcon     2          2               10
falcon     2          2               10
  fish      0          0                8
  dog       4          0                2
falcon     2          2               10
falcon     2          2               10
falcon     2          2               10
  fish      0          0                8
  dog       4          0                2
falcon     2          2               10
falcon     2          2               10
falcon     2          2               10
  fish      0          0                8
  dog       4          0                2
>>>
```

Extract 3 random elements from the Series df['num_legs']: Note that we use random_state to ensure the reproducibility of the examples.

```python
>>> df['num_legs'].sample(n=3, random_state=1)
fish  0
spider 8
falcon 2
Name: num_legs, dtype: int64
```

A random 50% sample of the DataFrame with replacement:

```python
>>> df.sample(frac=0.5, replace=True, random_state=1)
   num_legs  num_wings  num_specimen_seen
  dog     4          0               2
  fish     0          0               8
falcon   2          2               10
falcon   2          2               10
falcon   2          2               10
  fish     0          0               8
  dog     4          0               2
falcon   2          2               10
falcon   2          2               10
falcon   2          2               10
  fish     0          0               8
  dog     4          0               2
```

An upsample sample of the DataFrame with replacement: Note that replace parameter has to be True for frac parameter > 1.

```python
>>> df.sample(frac=2, replace=True, random_state=1)
   num_legs  num_wings  num_specimen_seen
  dog     4          0               2
  fish     0          0               8
falcon   2          2               10
falcon   2          2               10
falcon   2          2               10
  fish     0          0               8
  dog     4          0               2
falcon   2          2               10
falcon   2          2               10
falcon   2          2               10
  fish     0          0               8
  dog     4          0               2
```

3.3. Series

1375
Using a DataFrame column as weights. Rows with larger value in the `num_specimen_seen` column are more likely to be sampled.

```python
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
  num_legs  num_wings  num_specimen_seen
  falcon     2          2             10
  fish       0          0              8
```

### pandas.Series.searchsorted

`Series.searchsorted(value, side='left', sorter=None)`

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Series `self` such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

**Note:** The Series *must* be monotonically sorted, otherwise wrong locations will likely be returned. Pandas does not check this for you.

**Parameters**

- **value** [array-like or scalar] Values to insert into `self`.
- **side** [{`'left'`, `'right'`}, optional] If `'left'`, the index of the first suitable location found is given. If `'right'`, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).
- **sorter** [1-D array-like, optional] Optional array of integer indices that sort `self` into ascending order. They are typically the result of `np.argsort`.

**Returns**

- **int or array of int** A scalar or array of insertion points with the same shape as `value`.

**See also:**

- `sort_values` Sort by the values along either axis.
- `numpy.searchsorted` Similar method from NumPy.

**Notes**

Binary search is used to find the required insertion points.
Examples

```python
>>> ser = pd.Series([1, 2, 3])
>>> ser
0  1  
1  2  
2  3  
dtype: int64
```

```python
>>> ser.searchsorted(4)
3
```

```python
>>> ser.searchsorted([0, 4])
array([0, 3])
```

```python
>>> ser.searchsorted([1, 3], side='left')
array([0, 2])
```

```python
>>> ser.searchsorted([1, 3], side='right')
array([1, 3])
```

```python
>>> ser
0   2000-03-11
1   2000-03-12
2   2000-03-13
dtype: datetime64[ns]
```

```python
>>> ser.searchsorted('3/14/2000')
3
```

```python
>>> ser = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'], ordered=True)
>>> ser
['apple', 'bread', 'bread', 'cheese', 'milk']
Categories (4, object): ['apple' < 'bread' < 'cheese' < 'milk']
```

```python
>>> ser.searchsorted('bread')
1
```

```python
>>> ser.searchsorted(['bread'], side='right')
array([3])
```

If the values are not monotonically sorted, wrong locations may be returned:

```python
>>> ser = pd.Series([2, 1, 3])
>>> ser
0  2  
1  1  
```
(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> ser.searchsorted(1)
0  # wrong result, correct would be 1

pandas.Series.sem

Series.sem(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

- **axis** ([index (0)])
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

- **scalar** or **Series** (if level specified)

pandas.Series.set_axis

Series.set_axis(labels, axis=0, inplace=False)

Assign desired index to given axis.

Indexes for row labels can be changed by assigning a list-like or Index.

Parameters

- **labels** [list-like, Index] The values for the new index.
- **axis** [[0 or ‘index’], default 0] The axis to update. The value 0 identifies the rows.
- **inplace** [bool, default False] Whether to return a new Series instance.

Returns

- **renamed** [Series or None] An object of type Series or None if inplace=True.

See also:

- **Series.rename_axis** Alter the name of the index.
Examples

```python
>>> s = pd.Series([1, 2, 3])
>>> s
0   1
1   2
2   3
dtype: int64

>>> s.set_axis(['a', 'b', 'c'], axis=0)
a  1
b  2
c  3
dtype: int64
```

**pandas.Series.set_flags**

`Series.set_flags(*, copy=False, allows_duplicate_labels=None)`

Return a new object with updated flags.

**Parameters**

- `allows_duplicate_labels` [bool, optional] Whether the returned object allows duplicate labels.

**Returns**

- `Series` or `DataFrame` The same type as the caller.

**See also:**

- `DataFrame.attrs` Global metadata applying to this dataset.
- `DataFrame.flags` Global flags applying to this object.

**Notes**

This method returns a new object that’s a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

“Flags” differ from “metadata”. Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`. 
Examples

```python
>>> df = pd.DataFrame({'A': [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

`pandas.Series.shift`

`Series.shift(periods=1, freq=None, axis=0, fill_value=None)`

Shift index by desired number of periods with an optional time `freq`.

When `freq` is not passed, shift the index without realigning the data. If `freq` is passed (in this case, the index must be date or datetime, or it will raise a `NotImplementedError`), the index will be increased using the periods and the `freq`. `freq` can be inferred when specified as “infer” as long as either freq or inferred_freq attribute is set in the index.

**Parameters**

- **periods** [int] Number of periods to shift. Can be positive or negative.
- **freq** [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. ‘EOM’). If `freq` is specified then the index values are shifted but the data is not realigned. That is, use `freq` if you would like to extend the index when shifting and preserve the original data. If `freq` is specified as “infer” then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a `ValueError` is thrown.
- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default None] Shift direction.
- **fill_value** [object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of `self`. For numeric data, `np.nan` is used. For date-time, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype`. `na_value` is used.

*Changed in version 1.1.0.*

**Returns**

- **Series** Copy of input object, shifted.

**See also:**

- `Index.shift` Shift values of Index.
- `DatetimeIndex.shift` Shift values of DatetimeIndex.
- `PeriodIndex.shift` Shift values of PeriodIndex.
- `tshift` Shift the time index, using the index’s frequency if available.
Examples

```python
>>> df = pd.DataFrame(
    {
        "Col1": [10, 20, 15, 30, 45],
        "Col2": [13, 23, 18, 33, 48],
        "Col3": [17, 27, 22, 37, 52],
    },
    index=pd.date_range("2020-01-01", "2020-01-05")
)
>>> df
     Col1 Col2 Col3
2020-01-01  10   13  17
2020-01-02  20   23  27
2020-01-03  15   18  22
2020-01-04  30   33  37
2020-01-05  45   48  52
```

```python
>>> df.shift(periods=3)
     Col1 Col2 Col3
2020-01-01  NaN  NaN  NaN
2020-01-02  NaN  NaN  NaN
2020-01-03  NaN  NaN  NaN
2020-01-04  10.0  13.0  17.0
2020-01-05  20.0  23.0  27.0
```

```python
>>> df.shift(periods=1, axis="columns")
     Col1 Col2 Col3
2020-01-01  NaN  10   13
2020-01-02  NaN  20   23
2020-01-03  NaN  15   18
2020-01-04  NaN  30   33
2020-01-05  NaN  45   48
```

```python
>>> df.shift(periods=3, fill_value=0)
     Col1 Col2 Col3
2020-01-01   0    0    0
2020-01-02   0    0    0
2020-01-03   0    0    0
2020-01-04  10   13   17
2020-01-05  20   23   27
```

```python
>>> df.shift(periods=3, freq="D")
     Col1 Col2 Col3
2020-01-04  10   13   17
2020-01-05  20   23   27
2020-01-06  15   18   22
2020-01-07  30   33   37
2020-01-08  45   48   52
```

```python
>>> df.shift(periods=3, freq="infer")
     Col1 Col2 Col3
2020-01-04  10   13   17
2020-01-05  20   23   27
2020-01-06  15   18   22
```

(continues on next page)
pandas.Series.skew

Series.skew(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return unbiased skew over requested axis.

Parameters

axis [(index (0))] Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.slice_shift

Series.slice_shift(periods=1, axis=0)

Equivalent to shift without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Deprecated since version 1.2.0: slice_shift is deprecated, use DataFrame/Series.shift instead.

Parameters

periods [int] Number of periods to move, can be positive or negative.

Returns

shifted [same type as caller]

Notes

While the slice_shift is faster than shift, you may pay for it later during alignment.
pandas.Series.sort_index

Series.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)

Sort Series by index labels.

Returns a new Series sorted by label if inplace argument is False, otherwise updates the original series and returns None.

Parameters

- **axis** [int, default 0] Axis to direct sorting. This can only be 0 for Series.
- **level** [int, optional] If not None, sort on values in specified index level(s).
- **ascending** [bool or list-like of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.
- **inplace** [bool, default False] If True, perform operation in-place.
- **kind** [{‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’}, default ‘quicksort’] Choice of sorting algorithm. See also numpy.sort() for more information. ‘mergesort’ and ‘stable’ are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** [{‘first’, ‘last’}, default ‘last’] If ‘first’ puts NaNs at the beginning, ‘last’ puts NaNs at the end. Not implemented for MultiIndex.
- **sort_remaining** [bool, default True] If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.
- **ignore_index** [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.

New in version 1.0.0.

- **key** [callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the key argument in the built-in sorted() function, with the notable difference that this key function should be vectorized. It should expect an Index and return an Index of the same shape.

New in version 1.1.0.

Returns

- **Series or None** The original Series sorted by the labels or None if inplace=True.

See also:

- **DataFrame.sort_index** Sort DataFrame by the index.
- **DataFrame.sort_values** Sort DataFrame by the value.
- **Series.sort_values** Sort Series by the value.
Examples

```python
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, 4])
>>> s.sort_index()
1   c
2   b
3   a
4   d
dtype: object
```

Sort Descending

```python
>>> s.sort_index(ascending=False)
4   d
3   a
2   b
1   c
dtype: object
```

Sort Inplace

```python
>>> s.sort_index(inplace=True)
>>> s
1   c
2   b
3   a
4   d
dtype: object
```

By default NaNs are put at the end, but use `na_position` to place them at the beginning

```python
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, np.nan])
>>> s.sort_index(na_position='first')
NaN  d
   1.0  c
   2.0  b
   3.0  a
dtype: object
```

Specify index level to sort

```python
>>> arrays = [np.array(['qux', 'qux', 'foo', 'foo', 'baz', 'baz', 'bar', 'bar']),
            np.array(['two', 'one', 'two', 'one', 'two', 'one'])]
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
>>> s.sort_index(level=1)
bar  one  8
baz  one  6
foo  one  4
qux  one  2
bar  two  7
baz  two  5
foo  two  3
```

(continues on next page)
Does not sort by remaining levels when sorting by levels

```
>>> s.sort_index(level=1, sort_remaining=False)
qux one 2
foo one 4
baz one 6
bar one 8
qux two 1
foo two 3
baz two 5
bar two 7
dtype: int64
```

Apply a key function before sorting

```
>>> s = pd.Series([1, 2, 3, 4], index=['A', 'b', 'C', 'd'])
>>> s.sort_index(key=lambda x : x.str.lower())
A 1
b 2
C 3
d 4
dtype: int64
```

**pandas.Series.sort_values**

`Series.sort_values` is used to sort a Series in ascending or descending order by some criterion.

**Parameters**

- **axis** ([0 or ‘index’], default 0) Axis to direct sorting. The value ‘index’ is accepted for compatibility with DataFrame.sort_values.
- **ascending** [bool or list of bools, default True] If True, sort values in ascending order, otherwise descending.
- **inplace** [bool, default False] If True, perform operation in-place.
- **kind** ['quicksort', 'mergesort', 'heapsort', 'stable'], default ‘quicksort’ Choice of sorting algorithm. See also `numpy.sort()` for more information. ‘mergesort’ and ‘stable’ are the only stable algorithms.
- **na_position** [‘first’ or ‘last’, default ‘last’] Argument ‘first’ puts NaNs at the beginning, ‘last’ puts NaNs at the end.
- **ignore_index** [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.

New in version 1.0.0.
**key** [callable, optional] If not None, apply the key function to the series values before sorting. This is similar to the key argument in the builtin :func:`sorted()` function, with the notable difference that this key function should be *vectorized*. It should expect a :class:`Series` and return an array-like.

New in version 1.1.0.

**Returns**

*Series or None* Series ordered by values or None if :attr:`inplace=True`.

**See also:**

*Series.sort_index* Sort by the Series indices.

*DataFrame.sort_values* Sort DataFrame by the values along either axis.

*DataFrame.sort_index* Sort DataFrame by indices.

**Examples**

```python
>>> s = pd.Series([np.nan, 1, 3, 10, 5])
>>> s
0   NaN
1    1.0
2    3.0
3   10.0
4    5.0
dtype: float64

Sort values ascending order (default behaviour)

```python
>>> s.sort_values(ascending=True)
1    1.0
2    3.0
4    5.0
3   10.0
0   NaN
dtype: float64
```

Sort values descending order

```python
>>> s.sort_values(ascending=False)
3   10.0
4    5.0
2    3.0
1    1.0
0   NaN
dtype: float64
```

Sort values inplace

```python
>>> s.sort_values(ascending=False, inplace=True)
>>> s
3   10.0
4    5.0
```
Sort values putting NAs first

```python
>>> s.sort_values(na_position='first')
0   NaN
1    1.0
2    3.0
4    5.0
3   10.0
dtype: float64
```

Sort a series of strings

```python
>>> s = pd.Series(['z', 'b', 'd', 'a', 'c'])
```

```python
>>> s.sort_values()
3   a
1   b
4   c
2   d
0   z
dtype: object
```

Sort using a key function. Your `key` function will be given the `Series` of values and should return an array-like.

```python
>>> s = pd.Series(['a', 'B', 'c', 'D', 'e'])
```

```python
>>> s.sort_values()
1   B
3   D
0   a
2   c
4   e
dtype: object
```

```python
>>> s.sort_values(key=lambda x: x.str.lower())
0   a
1   B
2   c
3   D
4   e
dtype: object
```
NumPy ufuncs work well here. For example, we can sort by the \texttt{sin} of the value

```
>>> s = pd.Series([-4, -2, 0, 2, 4])
>>> s.sort_values(key=np.sin)
1   -2
4    4
2     0
0    -4
3    2
dtype: int64
```

More complicated user-defined functions can be used, as long as they expect a Series and return an array-like

```
>>> s.sort_values(key=lambda x: (np.tan(x.cumsum())))
0   -4
3    2
4    4
1   -2
2    0
dtype: int64
```

\texttt{pandas.Series.sparse}

\texttt{Series.sparse()}

Accessor for \texttt{SparseSeries} from other sparse matrix data types.

\texttt{pandas.Series.squeeze}

\texttt{Series.squeeze(axis=None)}

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don’t know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call \texttt{squeeze} to ensure you have a Series.

\textbf{Parameters}

- \texttt{axis} \{0 or ‘index’, 1 or ‘columns’, None\}, default None A specific axis to squeeze. By default, all length-1 axes are squeezed.

\textbf{Returns}

- \texttt{DataFrame, Series, or scalar} The projection after squeezing \texttt{axis} or all the axes.

\textbf{See also:}

- \texttt{Series.iloc} Integer-location based indexing for selecting scalars.
- \texttt{DataFrame.iloc} Integer-location based indexing for selecting Series.
- \texttt{Series.to_frame} Inverse of DataFrame.squeeze for a single-column DataFrame.
Examples

```python
>>> primes = pd.Series([2, 3, 5, 7])

Slicing might produce a Series with a single value:

```python
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0 2
dtype: int64
```  
```python
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```python
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1 3
2 5
3 7
dtype: int64
```  
```python
>>> odd_primes.squeeze()
1 3
2 5
3 7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0 1  2
1 3  4

Slicing a single column will produce a DataFrame with the columns having only one value:

```python
>>> df_a = df[['a']]  
>>> df_a
   a
0 1
1 3
```

So the columns can be squeezed down, resulting in a Series:

```python
>>> df_a.squeeze('columns')
0 1
1 3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```python
```
>>> df_0a = df.loc[df.index < 1, ['a']]

Squeezing the rows produces a single scalar Series:

```python
>>> df_0a.squeeze('rows')
a  1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```python
>>> df_0a.squeeze()
1
```

**pandas.Series.std**

Series.std(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)  
Return sample standard deviation over requested axis.  
Normalized by N-1 by default. This can be changed using the ddof argument.

**Parameters**

axis [{index (0)}]  
skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

scalar or Series (if level specified)

**Notes**

To have the same behaviour as numpy.std, use ddof=0 (instead of the default ddof=1)
### Examples

```python
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
                    'age': [21, 25, 62, 43],
                    'height': [1.61, 1.87, 1.49, 2.01]})
>>> df.set_index('person_id')
               age  height
person_id
0         21     1.61
1         25     1.87
2         62     1.49
3         43     2.01
```

The standard deviation of the columns can be found as follows:

```python
>>> df.std()
    age     height
age  18.786076  0.237417
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:

```python
>>> df.std(ddof=0)
    age     height
age  16.269219  0.205609
```

### pandas.Series.str

**Series.str()**

Vectorized string functions for Series and Index.

NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

#### Examples

```python
>>> s = pd.Series(['A_Str_Series'])
```

```python
>>> s
0  A_Str_Series
dtype: object
```

```python
>>> s.str.split('_')
0  ['A', 'Str', 'Series']
dtype: object
```

```python
>>> s.str.replace('_', '')
0  AStrSeries
dtype: object
```
pandas.Series.sub

Series.sub(other, level=None, fill_value=None, axis=0)

Return Subtraction of series and other, element-wise (binary operator sub).

Equivalent to series - other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns

Series The result of the operation.

See also:

Series.rsub Reverse of the Subtraction operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.subtract(b, fill_value=0)
a 0.0
b 1.0
c 1.0
d -1.0
e NaN
dtype: float64
```
pandas.Series.subtract

Series.subtract(other, level=None, fill_value=None, axis=0)

Return Subtraction of series and other, element-wise (binary operator sub).

Equivalent to series - other, but with support to substitute a fill_value for missing data in either one of the inputs.

Parameters

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

See also:

Series.rsub Reverse of the Subtraction operator, see Python documentation for more details.

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
>>> a.subtract(b, fill_value=0)
a  0.0
b  1.0
c  1.0
d -1.0
e  NaN
dtype: float64
```
pandas.Series.sum

Series.sum(axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs)

Return the sum of the values over the requested axis.

This is equivalent to the method numpy.sum.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**kwargs Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.sum Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
def main():
    idx = pd.MultiIndex.from_arrays([
        ['warm', 'warm', 'cold', 'cold'],
        ['dog', 'falcon', 'fish', 'spider']
    ], names=['blooded', 'animal'])
    s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
    print(s.blooded)
```
spider 8
Name: legs, dtype: int64

```python
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is 0.

```python
>>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you’d like the sum of an empty series to be NaN, pass `min_count=1`.

```python
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).sum()
0.0
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

### pandas.Series.swapaxes

`Series.swapaxes(axis1, axis2, copy=True)`

Interchange axes and swap values axes appropriately.

Returns

- `y` [same as input]

### pandas.Series.swaplevel

`Series.swaplevel(i=-2, j=-1, copy=True)`

Swap levels i and j in a `MultiIndex`.

Default is to swap the two innermost levels of the index.

Parameters

- `i, j` [int or str] Levels of the indices to be swapped. Can pass level name as string.
- `copy` [bool, default True] Whether to copy underlying data.

Returns

- `Series` Series with levels swapped in MultiIndex.
Examples

```python
>>> s = pd.Series(
    ...     ['A', 'B', 'A', 'C'],
    ...     index=[
        ...         ['Final exam', 'Final exam', 'Coursework', 'Coursework'],
        ...         ['History', 'Geography', 'History', 'Geography'],
        ...         ['January', 'February', 'March', 'April'],
    ...     ],
    ... )
>>> s
Final exam History January A
   Geography February B
Coursework History March A
   Geography April C
dtype: object
```

In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```python
>>> s.swaplevel()
Final exam January History A
    February Geography B
Coursework March History A
    April Geography C
dtype: object
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```python
>>> s.swaplevel(0)
January History Final exam A
    February Geography Final exam B
Coursework March Coursework A
    April Coursework C
dtype: object
```

We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```python
>>> s.swaplevel(0, 1)
History Final exam January A
    Geography Final exam February B
Coursework March Coursework A
    Geography Coursework April C
dtype: object
```
Series.tail

Series.tail(n=5)

Return the last n rows.

This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of n, this function returns all rows except the first n rows, equivalent to df[n:]

Parameters

n [int, default 5] Number of rows to select.

Returns

The last n rows of the caller object.

See also:

DataFrame.head The first n rows of the caller object.

Examples

```python
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion', 'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
animal
0 alligator
1 bee
2 falcon
3 lion
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

Viewing the last 5 lines

```python
>>> df.tail()
animal
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

Viewing the last n lines (three in this case)

```python
>>> df.tail(3)
animal
6 shark
7 whale
8 zebra
```
For negative values of \( n \):

```python
>>> df.tail(-3)
animal
3  lion
4  monkey
5  parrot
6  shark
7  whale
8  zebra
```

### pandas.Series.take

**Series.take\((indices, axis=0, is_copy=None, **kwargs)\)**

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters**

- **indices** [array-like] An array of ints indicating which positions to take.
- **axis** [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements.
  - 0 means that we are selecting rows,
  - 1 means that we are selecting columns.
- **is_copy** [bool] Before pandas 1.0, `is_copy=False` can be specified to ensure that the return value is an actual copy. Starting with pandas 1.0, `take` always returns a copy, and the keyword is therefore deprecated.
  
  Deprecated since version 1.0.0.

- **kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

**Returns**

- **taken** [same type as caller] An array-like containing the elements taken from the object.

**See also:**

- `DataFrame.loc` Select a subset of a DataFrame by labels.
- `DataFrame.iloc` Select a subset of a DataFrame by positions.
- `numpy.take` Take elements from an array along an axis.

**Examples**

```python
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=['name', 'class', 'max_speed'],
...                     index=[0, 2, 3, 1])
```

(continues on next page)
Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That’s because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```python
>>> df.take([0, 3])
    name   class  max_speed
0  falcon   bird     389.0
1  monkey  mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```python
>>> df.take([1, 2], axis=1)
   class  max_speed
0   bird     389.0
2   bird      24.0
3  mammal      80.5
1  mammal      NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```python
>>> df.take([-1, -2])
    name   class  max_speed
1  monkey  mammal      NaN
3   lion  mammal      80.5
```

---

**pandas.Series.to_clipboard**

Series.to_clipboard(excel=True, sep=None, **kwargs)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

**Parameters**

- **excel** [bool, default True] Produce output in a csv format for easy pasting into excel.
  - True, use the provided separator for csv pasting.
  - False, write a string representation of the object to the clipboard.
- **sep** [str, default \'\t\'] Field delimiter.
- **kwargs** These parameters will be passed to DataFrame.to_csv.

See also:

DataFrame.to_csv Write a DataFrame to a comma-separated values (csv) file.
read_clipboard Read text from clipboard and pass to read_csv.
Notes

Requirements for your platform.

- Linux: `xclip` or `xsel` (with `PyQt4` modules)
- Windows: none
- macOS: none

Examples

Copy the contents of a DataFrame to the clipboard.

```python
def = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
...
# Wrote the following to the system clipboard:
... # A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword `index` and setting it to false.

```python
>>> df.to_clipboard(sep=',', index=False)
...
# Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

`pandas.Series.to_csv`

`Series.to_csv`

Write object to a comma-separated values (csv) file.

**Parameters**

- `path_or_buf` [str, path object, file-like object, or None, default None] String, path object (implementing `os.PathLike[str]`), or file-like object implementing a `write()` function. If `None`, the result is returned as a string. If a non-binary file object is passed, it should be opened with `newline='\n'`, disabling universal newlines. If a binary file object is passed, `mode` might need to contain a `b`.

  Changed in version 1.2.0: Support for binary file objects was introduced.

- `sep` [str, default ' '] String of length 1. Field delimiter for the output file.

- `na_rep` [str, default ''] Missing data representation.

- `float_format` [str, default None] Format string for floating point numbers.

- `columns` [sequence, optional] Columns to write.
header [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

index_label [str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and header and index are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode [str] Python write mode, default ‘w’.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to ‘utf-8’. encoding is not supported if path_or_buf is a non-binary file object.

compression [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘%s’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

quoting [optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a float_format then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar [str, default ‘”’] String of length 1. Character used to quote fields.

line_terminator [str, optional] The newline character or character sequence to use in the output file. Defaults to os.linesep, which depends on the OS in which this method is called (‘\n’ for linux, ‘\rn’ for Windows, i.e.).

chunksize [int or None] Rows to write at a time.

date_format [str, default None] Format string for datetime objects.

doublequote [bool, default True] Control quoting of quotechar inside a field.

escapechar [str, default None] String of length 1. Character used to escape sep and quotechar when appropriate.

decimal [str, default ‘.’] Character recognized as decimal separator. E.g. use ‘,’ for European data.

errors [str, default ‘strict’] Specifies how encoding and decoding errors are to be handled. See the errors argument for open() for a full list of options.

New in version 1.1.0.
**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

**Returns**

None or str  If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

*read_csv*  Load a CSV file into a DataFrame.

*to_excel*  Write DataFrame to an Excel file.

**Examples**

```python
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
name,mask,weapon
Raphael,red,sai
Donatello,purple,bo staff

Create ‘out.zip’ containing ‘out.csv’

```python
>>> compression_opts = dict(method='zip',
...                           archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```
**pandas.Series.to_dict**

`Series.to_dict(into=<class 'dict'>)`

Convert Series to `{label -> value}` dict or dict-like object.

**Parameters**

- `into` [class, default dict] The `collections.abc.Mapping` subclass to use as the return object. Can be the actual class or an empty instance of the mapping type you want. If you want a `collections.defaultdict`, you must pass it initialized.

**Returns**

- `collections.abc.Mapping` Key-value representation of Series.

**Examples**

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_dict()
{0: 1, 1: 2, 2: 3, 3: 4}
>>> from collections import OrderedDict, defaultdict
>>> s.to_dict(OrderedDict)
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> dd = defaultdict(list)
>>> s.to_dict(dd)
defaultdict(<class 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})
```

**pandas.Series.to_excel**

`Series.to_excel(excel_writer, sheet_name='Sheet1', na_rep='.', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options=None)`

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an `ExcelWriter` object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique `sheet_name`. With all data written to the file it is necessary to save the changes. Note that creating an `ExcelWriter` object with a file name that already exists will result in the contents of the existing file being erased.

**Parameters**

- `excel_writer` [path-like, file-like, or ExcelWriter object] File path or existing ExcelWriter.
- `sheet_name` [str, default ‘Sheet1’] Name of sheet which will contain DataFrame.
- `na_rep` [str, default ‘.’] Missing data representation.
- `float_format` [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.
- `columns` [sequence or list of str, optional] Columns to write.
**header** [bool or list of str, default True] Write out the column names. If a list of string is
given it is assumed to be aliases for the column names.

**index** [bool, default True] Write row names (index).

**index_label** [str or sequence, optional] Column label for index column(s) if desired. If not
specified, and **header** and **index** are True, then the index names are used. A sequence
should be given if the DataFrame uses MultiIndex.

**startrow** [int, default 0] Upper left cell row to dump data frame.

**startcol** [int, default 0] Upper left cell column to dump data frame.

**engine** [str, optional] Write engine to use, ‘openpyxl’ or ‘xlsxwriter’. You can also set this
Deprecation since version 1.2.0: As the xlwt package is no longer maintained, the xlwt
engine will be removed in a future version of pandas.

**merge_cells** [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

**encoding** [str, optional] Encoding of the resulting excel file. Only necessary for xlwt,
other writers support unicode natively.

**inf_rep** [str, default ‘inf’] Representation for infinity (there is no native representation for
infinity in Excel).

**verbose** [bool, default True] Display more information in the error logs.

**freeze_panes** [tuple of int (length 2), optional] Specifies the one-based bottommost row
and rightmost column that is to be frozen.

**storage_options** [dict, optional] Extra options that make sense for a particular storage
connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-
value pairs are forwarded to urllib as header options. For other URLs (e.g. starting
with “s3://”, “gcs://”) the key-value pairs are forwarded to fsspec. Please see
fsspec and urllib for more details.

New in version 1.2.0.

**See also:**

to_csv  Write DataFrame to a comma-separated values (csv) file.

ExcelWriter  Class for writing DataFrame objects into excel sheets.

read_excel  Read an Excel file into a pandas DataFrame.

read_csv  Read a comma-separated values (csv) file into DataFrame.

**Notes**

For compatibility with to_csv(), to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible to write further data without rewriting the whole work-
book.
**Examples**

Create, write to and save a workbook:

```python
>>> df1 = pd.DataFrame([["a", "b"], ["c", "d"]],
    index=['row 1', 'row 2'],
    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```python
>>> df1.to_excel("output.xlsx",
    sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```python
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet_name_1')
    df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```python
>>> with pd.ExcelWriter('output.xlsx',
    mode='a') as writer:
    df.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the `engine` keyword (the default engine is automatically chosen depending on the file extension):

```python
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

---

**pandas.Series.to_frame**

`Series.to_frame(name=NoDefault.no_default)`

Convert Series to DataFrame.

**Parameters**

- **name** [object, optional] The passed name should substitute for the series name (if it has one).

**Returns**

- **DataFrame** DataFrame representation of Series.
Examples

```python
>>> s = pd.Series(['a', 'b', 'c'],
                name='vals')
>>> s.to_frame()
   vals
0   a
1   b
2   c
```

`pandas.Series.to_hdf`

`Series.to_hdf(path_or_buf, key, mode='a', complevel=None, complib=None, append=False,
               format=None, index=True, min_itemsize=None, nan_rep=None, dropna=None,
               data_columns=None, errors='strict', encoding='UTF-8')`

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different key.

**Warning:** One can store a subclass of DataFrame or Series to HDF5, but the type of the subclass is lost upon storing.

For more information see the *user guide*.

**Parameters**

- **path_or_buf** [str or pandas.HDFStore] File path or HDFStore object.
- **key** [str] Identifier for the group in the store.
- **mode** [{'a', 'w', 'r+'}, default 'a'] Mode to open file:
  - 'w': write, a new file is created (an existing file with the same name would be deleted).
  - 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
  - 'r+': similar to 'a', but the file must already exist.
- **complevel** [{0-9}, default None] Specifies a compression level for data. A value of 0 or None disables compression.
- **complib** [{'zlib', 'lzma', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: ‘blosc:blosclz’): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.
- **append** [bool, default False] For Table formats, append the input data to the existing.
- **format** [{'fixed', 'table', None}, default 'fixed'] Possible values:

• ‘table’: Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

• If None, pd.get_option(‘io.hdf.default_format’) is checked, followed by fallback to “fixed”.

errors [str, default ‘strict’] Specifies how encoding and decoding errors are to be handled. See the errors argument for open() for a full list of options.

encoding [str, default “UTF-8”]

min_itemsize [dict or int, optional] Map column names to minimum string sizes for columns.


data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See Query via data columns. Applicable only to format=’table’.

See also:

read_hdf Read from HDF file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

DataFrame.to_sql Write to a SQL table.

DataFrame.to_feather Write out feather-format for DataFrames.

DataFrame.to_csv Write out to a csv file.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
    index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```python
>>> pd.read_hdf('data.h5', 'df')
A   B
a  1   4
b  2   5
c  3   6
```

```python
>>> pd.read_hdf('data.h5', 's')
0   1
1   2
2   3
3   4
dtype: int64
```
Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters**

- **path_or_buf** [str, path object, file-like object, or None, default None] String, path object (implementing os.PathLike[bool]), or file-like object implementing a write() function. If None, the result is returned as a string.

- **orient** [str] Indication of expected JSON string format.
  - Series:
    - default is 'index'
    - allowed values are: {'split', 'records', 'index', 'table'}.
  - DataFrame:
    - default is 'columns'
    - allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.
  - The format of the JSON string:
    - 'split': dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
    - 'records': list like [[column -> value], ... , [column -> value]]
    - 'index': dict like {index -> {column -> value}}
    - 'columns': dict like {column -> {index -> value}}
    - 'values': just the values array
    - 'table': dict like {'schema': {schema}, 'data': {data}}

Describing the data, where data component is like orient='records'.

- **date_format** ['epoch', 'iso'] Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the orient. For orient='table', the default is 'iso'. For all other orients, the default is 'epoch'.

- **double_precision** [int, default 10] The number of decimal places to use when encoding floating point values.

- **force_ascii** [bool, default True] Force encoded string to be ASCII.

- **date_unit** ['s', 'ms', 'us', 'ns'] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

- **default_handler** [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

- **lines** [bool, default False] If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.
compression [str or dict, default ‘infer’] For on-the-fly compression of the output data. If ‘infer’ and ‘path_or_buf’ path-like, then detect compression from the following extensions: ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, or ‘.zst’ (otherwise no compression). Set to None for no compression. Can also be a dict with key ‘method’ set to one of {‘zip’, ‘gzip’, ‘bz2’, ‘zstd’} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, ‘mtime’: 1}.

Changed in version 1.4.0: Zstandard support.

index [bool, default True] Whether to include the index values in the JSON string. Not including the index (index=False) is only supported when orient is ‘split’ or ‘table’.

indent [int, optional] Length of whitespace used to indent each record.

New in version 1.0.0.

storage_options [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

Returns

None or str If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.

See also:

read_json Convert a JSON string to pandas object.

Notes

The behavior of indent=0 varies from the stdlib, which does not indent the output but does insert newlines. Currently, indent=0 and the default indent=None are equivalent in pandas, though this may change in a future release.

orient='table' contains a ‘pandas_version’ field under ‘schema’. This stores the version of pandas used in the latest revision of the schema.

Examples

```python
>>> import json
>>> df = pd.DataFrame([[
"a", "b"], ["c", "d"],
... index=["row 1", "row 2"],
... columns=["col 1", "col 2"]
... ])
```
Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding:

```python
>>> result = df.to_json(orient="records")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
[{
    "col 1": "a",
    "col 2": "b"
},
{
    "col 1": "c",
    "col 2": "d"
}]
```

Encoding/decoding a Dataframe using 'index' formatted JSON:

```python
>>> result = df.to_json(orient="index")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
    }
}  

(continues on next page)
Encoding/decoding a Dataframe using 'columns' formatted JSON:

```python
>>> result = df.to_json(orient="columns")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
  "col 1": {
    "row 1": "a",
    "row 2": "c"
  },
  "col 2": {
    "row 1": "b",
    "row 2": "d"
  }
}
```

Encoding/decoding a Dataframe using 'values' formatted JSON:

```python
>>> result = df.to_json(orient="values")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
[
  [
    "a",
    "b"
  ],
  [
    "c",
    "d"
  ]
]
```

Encoding with Table Schema:

```python
>>> result = df.to_json(orient="table")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{  
  "schema": {
    "fields": [
      {
        "name": "index",
        "type": "string"
      },
      {
        "name": "col 1",
        "type": "string"
      },
      {
        "name": "col 2",
        "type": "string"
      }
    ]
  }
}
```
pandas.Series.to_latex

Series.to_latex(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None, decimal='', multicolumn=None, multicolumn_format=None, multirow=None, caption=None, label=None, position=None)

Render object to a LaTeX tabular, longtable, or nested table.

Requires \usepackage{booktabs}. The output can be copy/pasted into a main LaTeX document or read from an external file with \input{table.tex}.

Changed in version 1.0.0: Added caption and label arguments.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

**Parameters**

- **buf** [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.
- **columns** [list of label, optional] The subset of columns to write. Writes all columns by default.
- **col_space** [int, optional] The minimum width of each column.
- **header** [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.
- **index** [bool, default True] Write row names (index).
- **na_rep** [str, default ‘NaN’] Missing data representation.
formatters [list of functions or dict of [str: function], optional] Formatter functions to apply to columns’ elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function or str, optional, default None] Formatter for floating point numbers. For example float_format="%.2f" and float_format="{:0.2f}". format will both result in 0.1234 being formatted as 0.12.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names [bool, default True] Prints the names of the indexes.

bold_rows [bool, default False] Make the row labels bold in the output.

column_format [str, optional] The columns format as specified in LaTeX table format e.g. ‘rcI’ for 3 columns. By default, ‘l’ will be used for all columns except columns of numbers, which default to ‘r’.

longtable [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a usepackage{longtable} to your LaTeX preamble.

escape [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to ‘utf-8’.

decimal [str, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.

multicolumn [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

multicolumn_format [str, default ‘l’] The alignment for multicolumns, similar to column_format. The default will be read from the config module.

multirow [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

caption [str or tuple, optional] Tuple (full_caption, short_caption), which results in \caption[short_caption]{full_caption}; if a single string is passed, no short caption will be set.

New in version 1.0.0.

Changed in version 1.2.0: Optionally allow caption to be a tuple (full_caption, short_caption).

label [str, optional] The LaTeX label to be placed inside \label{} in the output. This is used with \ref{} in the main .tex file.

New in version 1.0.0.

position [str, optional] The LaTeX positional argument for tables, to be placed after \begin{} in the output.

New in version 1.2.0.

Returns
str or None  If buf is None, returns the result as a string. Otherwise returns None.

See also:

Styler.to_latex  Render a DataFrame to LaTeX with conditional formatting.

DataFrame.to_string  Render a DataFrame to a console-friendly tabular output.

DataFrame.to_html  Render a DataFrame as an HTML table.

Examples

```python
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
                        ...          mask=['red', 'purple'],
                        ...          weapon=['sai', 'bo staff']))
>>> print(df.to_latex(index=False))
\begin{tabular}{lll}
\toprule
name & mask & weapon \\
\midrule
Raphael & red & sai \\
Donatello & purple & bo staff \\
\bottomrule
\end{tabular}
```

pandas.Series.to_list

Series.to_list()  
Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

Returns  
list

See also:

numpy.ndarray.tolist  Return the array as an a.ndim-levels deep nested list of Python scalars.

pandas.Series.to_markdown

Series.to_markdown(buf=None, mode='wt', index=True, storage_options=None, **kwargs)  
Print Series in Markdown-friendly format.

New in version 1.0.0.

Parameters

buf  [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

mode  [str, optional] Mode in which file is opened, “wt” by default.
**index**  [bool, optional, default True] Add index (row) labels.

New in version 1.1.0.

**storage_options**  [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

**kwargs** These parameters will be passed to tabulate.

**Returns**

*str*  Series in Markdown-friendly format.

**Notes**

Requires the tabulate package.

**Examples**

```python
>>> s = pd.Series(["elk", "pig", "dog", "quetzal"], name="animal")
>>> print(s.to_markdown())
<table>
<thead>
<tr>
<th align="right"></th>
<th align="left">animal</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">0</td>
<td align="left">elk</td>
</tr>
<tr>
<td align="right">1</td>
<td align="left">pig</td>
</tr>
<tr>
<td align="right">2</td>
<td align="left">dog</td>
</tr>
<tr>
<td align="right">3</td>
<td align="left">quetzal</td>
</tr>
</tbody>
</table>
```

Output markdown with a tabulate option.

```python
>>> print(s.to_markdown(tablefmt="grid"))
+----+----------+
| | animal |
+====+==========+
| 0 | elk |
+----+----------+
| 1 | pig |
+----+----------+
| 2 | dog |
+----+----------+
| 3 | quetzal |
+----+----------+
```
pandas.Series.to_numpy

Series.to_numpy(dtype=None, copy=False, na_value=NoDefault.no_default, **kwargs)

A NumPy ndarray representing the values in this Series or Index.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to numpy.asarray().
copy [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that copy=False does not ensure that to_numpy() is no-copy. Rather, copy=True ensure that a copy is made, even if not strictly necessary.
na_value [Any, optional] The value to use for missing values. The default value depends on dtype and the type of the array.

New in version 1.0.0.

**kwargs Additional keywords passed through to the to_numpy method of the underlying array (for extension arrays).

New in version 1.0.0.

Returns

numpy.ndarray

See also:

Series.array Get the actual data stored within.
Index.array Get the actual data stored within.
DataFrame.to_numpy Similar method for DataFrame.

Notes

The returned array will be the same up to equality (values equal in self will be equal in the returned array; likewise for values that are not equal). When self contains an ExtensionArray, the dtype may be different. For example, for a category-dtype Series, to_numpy() will return a NumPy array and the categorical dtype will be lost.

For NumPy dtypes, this will be a reference to the actual data stored in this Series or Index (assuming copy=False). Modifying the result in place will modify the data stored in the Series or Index (not that we recommend doing that).

For extension types, to_numpy() may require copying data and coercing the result to a NumPy type (possibly object), which may be expensive. When you need a no-copy reference to the underlying data, Series.array should be used instead.

This table lays out the different dtypes and default return types of to_numpy() for various dtypes within pandas.

<table>
<thead>
<tr>
<th>dtype</th>
<th>array type</th>
</tr>
</thead>
<tbody>
<tr>
<td>category[T]</td>
<td>ndarray[T] (same dtype as input)</td>
</tr>
<tr>
<td>period</td>
<td>ndarray[object] (Periods)</td>
</tr>
<tr>
<td>interval</td>
<td>ndarray[object] (Intervals)</td>
</tr>
<tr>
<td>IntegerNA</td>
<td>ndarray[object]</td>
</tr>
<tr>
<td>datetime64[ns]</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>datetime64[ns, tz]</td>
<td>ndarray[object] (Timestamps)</td>
</tr>
</tbody>
</table>
Examples

```python
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

Specify the `dtype` to control how datetime-aware data is represented. Use `dtype=object` to return an 
ndarray of pandas `Timestamp` objects, each with the correct tz.

```python
>>> ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> ser.to_numpy(dtype=object)
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET')],
       dtype=object)
```

Or `dtype='datetime64[ns]'` to return an ndarray of native datetime64 values. The values are converted 
to UTC and the timezone info is dropped.

```python
>>> ser.to_numpy(dtype="datetime64[ns]"
... array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00...'],
       dtype='datetime64[ns]')
```

### pandas.Series.to_period

`Series.to_period(freq=None, copy=True)`

Convert Series from DatetimeIndex to PeriodIndex.

**Parameters**

- **freq** [str, default None] Frequency associated with the PeriodIndex.
- **copy** [bool, default True] Whether or not to return a copy.

**Returns**

- **Series** Series with index converted to PeriodIndex.

### pandas.Series.to_pickle

`Series.to_pickle(path, compression='infer', protocol=5, storage_options=None)`

Pickle (serialize) object to file.

**Parameters**

- **path** [str] File path where the pickled object will be stored.
- **compression** [str or dict, default 'infer'] For on-the-fly compression of the output data. 
  If 'infer' and 'path' path-like, then detect compression from the following extensions: 
  `.gz`, `.bz2`, `.zip`, `.xz`, or `.zst` (otherwise no compression). Set to None 
  for no compression. Can also be a dict with key 'method' set to one of {'zip', 
  'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile. 
  ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor, 
  respectively. As an example, the following could be passed for faster compression
and to create a reproducible gzip archive: `compression={'method': 'gzip',
'compresslevel': 1, 'mtime': 1}.

**protocol** [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1] paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

**storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to urllib as header options. For other URLs (e.g. starting with “s3://”, and “gcs://”) the key-value pairs are forwarded to fsspec. Please see fsspec and urllib for more details.

New in version 1.2.0.

See also:

**read_pickle** Load pickled pandas object (or any object) from file.

**DataFrame.to_hdf** Write DataFrame to an HDF5 file.

**DataFrame.to_sql** Write DataFrame to a SQL database.

**DataFrame.to_parquet** Write a DataFrame to the binary parquet format.

**Examples**

```python
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0   0   5
1   1   6
2   2   7
3   3   8
4   4   9
>>> original_df.to_pickle("./dummy.pkl")

>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0   0   5
1   1   6
2   2   7
3   3   8
4   4   9
```
pandas.Series.to_sql

Series.to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name [str] Name of SQL table.

con [sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable See here.

schema [str, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [['fail', 'replace', 'append'], default 'fail'] How to behave if the table already exists.

• fail: Raise a ValueError.
• replace: Drop the table before inserting new values.
• append: Insert new values to the existing table.

index [bool, default True] Write DataFrame index as a column. Uses index_label as the column name in the table.

index_label [str or sequence, default None] Column label for index column(s). If None is given (default) and index is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype [dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method [[None, ‘multi’, callable], optional] Controls the SQL insertion clause used:

• None: Uses standard SQL INSERT clause (one per row).
• ‘multi’: Pass multiple values in a single INSERT clause.
• callable with signature (pd_table, conn, keys, data_iter).

Details and a sample callable implementation can be found in the section insert method.

Returns

None or int Number of rows affected by to_sql. None is returned if the callable passed into method does not return the number of rows.

The number of returned rows affected is the sum of the rowcount attribute of sqlite3.Cursor or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.
New in version 1.4.0.

**Raises**

*ValueError* When the table already exists and *if_exists* is ‘fail’ (the default).

**See also:**

`read_sql` Read a DataFrame from a table.

**Notes**

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

**References**

[1], [2]

**Examples**

Create an in-memory SQLite database.

```python
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```python
>>> df = pd.DataFrame({'name': ['User 1', 'User 2', 'User 3']})
>>> df
    name
0  User 1
1  User 2
2  User 3
>>> df.to_sql('users', con=engine)
3
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An `sqlalchemy.engine.Connection` can also be passed to `con`:

```python
>>> with engine.begin() as connection:
...    df1 = pd.DataFrame({'name': ['User 4', 'User 5']})
...    df1.to_sql('users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.
Overwrite the table with just `df2`.

```python
>>> df2.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
2
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```python
>>> df = pd.DataFrame({'A': [1, None, 2]})
>>> df
  A
0  1.0
1  NaN
2  2.0

>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={'A': Integer()})
3
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

**pandas.Series.to_string**

`Series.to_string(buf=None, na_rep='NaN', float_format=None, header=True, index=True, length=False, dtype=False, name=False, max_rows=None, min_rows=None)`

Render a string representation of the Series.

**Parameters**

- `buf` [StringIO-like, optional] Buffer to write to.
- `na_rep` [str, optional] String representation of NaN to use, default 'NaN'.
- `float_format` [one-parameter function, optional] Formatter function to apply to columns' elements if they are floats, default None.
- `header` [bool, default True] Add the Series header (index name).
- `index` [bool, optional] Add index (row) labels, default True.
- `length` [bool, default False] Add the Series length.
**dtype** [bool, default False] Add the Series dtype.

**name** [bool, default False] Add the Series name if not None.

**max_rows** [int, optional] Maximum number of rows to show before truncating. If None, show all.

**min_rows** [int, optional] The number of rows to display in a truncated repr (when number of rows is above max_rows).

**Returns**

*str or None* String representation of Series if buf=None, otherwise None.

---

**pandas.Series.to_timestamp**

Series.to_timestamp(freq=None, how='start', copy=True)

Cast to DatetimeIndex of Timestamps, at beginning of period.

**Parameters**

*freq* [str, default frequency of PeriodIndex] Desired frequency.

*how* [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.

*copy* [bool, default True] Whether or not to return a copy.

**Returns**

*Series with DatetimeIndex*

---

**pandas.Series.to_xarray**

Series.to_xarray()

Return an xarray object from the pandas object.

**Returns**

*xarray.DataArray or xarray.Dataset* Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

**See also:**

*DataFrame.to_hdf* Write DataFrame to an HDF5 file.

*DataFrame.to_parquet* Write a DataFrame to the binary parquet format.

**Notes**

See the xarray docs
Examples

```python
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                             'num_legs'])
>>> df
name    class     max_speed  num_legs
0  falcon   bird       389.0      2
1  parrot   bird        24.0      2
2  lion    mammal      80.5       4
3  monkey  mammal     NaN        4
```

```python
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 4)
Coordinates:
* index   (index) int64 0 1 2 3
Data variables:
    name   (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class  (index) object 'bird' 'bird' 'mammal' 'mammal'
     max_speed   (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```python
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389., 24., 80.5, nan])
Coordinates:
* index   (index) int64 0 1 2 3
```

```python
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                                'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                                'speed': [350, 18, 361, 15]})
>>> df_multiindex
```

```python
>>> df-multiindex = df_multiindex.set_index(['date', 'animal'])
```

```python
>>> df_multiindex
speed
animal date
falcon 2018-01-01 350
      2018-01-02 361
parrot 2018-01-01 18
        2018-01-02 15
```

```python
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
... (continues on next page)
```
* date (date) datetime64[ns] 2018-01-01 2018-01-02
* animal (animal) object 'falcon' 'parrot'

Data variables:
  speed (date, animal) int64 350 18 361 15

**pandas.Series.tolist**

`Series.tolist()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

Returns

list

See also:

*numpy.ndarray.tolist* Return the array as an a.ndim-levels deep nested list of Python scalars.

**pandas.Series.transform**

`Series.transform(func, axis=0, *args, **kwargs)`

Call func on self producing a Series with the same axis shape as self.

Parameters

func [function, str, list-like or dict-like] Function to use for transforming the data. If a function, must either work when passed a Series or when passed to Series.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.

Accepted combinations are:

- function
- string function name
- list-like of functions and/or function names, e.g. [np.exp, 'sqrt']
- dict-like of axis labels -> functions, function names or list-like of such.

axis [[0 or 'index']] Parameter needed for compatibility with DataFrame.

*args Positional arguments to pass to func.

**kwargs Keyword arguments to pass to func.

Returns

Series A Series that must have the same length as self.

Raises

ValueError [If the returned Series has a different length than self.]

See also:

*Series.agg* Only perform aggregating type operations.
Series.apply  Invoke function on a Series.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```python
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3

>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting Series must have the same length as the input Series, it is possible to provide several input functions:

```python
>>> s = pd.Series(range(3))
>>> s
0  0  
1  1  
2  2  
dtype: int64

>>> s.transform([np.sqrt, np.exp])
       sqrt       exp
0  0.000000  1.000000
1  1.000000  2.718282
2  1.414214  7.389056
```

You can call transform on a GroupBy object:

```python
>>> df = pd.DataFrame({
...    "Date": ["2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
...    "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
>>> df
       Date  Data
0  2015-05-08    5
1  2015-05-07    8
2  2015-05-06    6
3  2015-05-05    1
4  2015-05-08   50
5  2015-05-07  100
```

(continues on next page)
>>> df.groupby('Date')['Data'].transform('sum')
0          55
1         108
2           66
3         121
4          55
5         108
6           66
7         121
Name: Data, dtype: int64

>>> df = pd.DataFrame({
...     "c": [1, 1, 1, 2, 2, 2, 2],
...     "type": ["m", "n", "o", "m", "m", "n", "n"]
... })
>>> df
   c  type
0  1    m
1  1    n
2  1    o
3  2    m
4  2    m
5  2    n
6  2    n

>>> df['size'] = df.groupby('c')['type'].transform(len)
>>> df
   c  type  size
0  1    m     3
1  1    n     3
2  1    o     3
3  2    m     4
4  2    m     4
5  2    n     4
6  2    n     4

pandas.Series.transpose

Series.transpose(*args, **kwargs)

Return the transpose, which is by definition self.

Returns

%(klass)s
**pandas.Series.truediv**

Series.truediv(other, level=None, fill_value=None, axis=0)

Return Floating division of series and other, element-wise (binary operator truediv).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in either one of the inputs.

**Parameters**

- **other** [Series or scalar value]
- **fill_value** [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result of filling (at that location) will be missing.

- **level** [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

Series The result of the operation.

**See also:**

`Series.rtruediv` Reverse of the Floating division operator, see Python documentation for more details.

**Examples**

```python
gf >>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
gf >>> a
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
gf >>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
gf >>> b
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
gf >>> a.divide(b, fill_value=0)
  a   1.0
  b  inf
  c  inf
  d   0.0
  e  NaN
dtype: float64
```
pandas.Series.truncate

Series.truncate(before=None, after=None, axis=None, copy=True)  
Truncate a Series or DataFrame before and after some index value.  
This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters  
- **before** [date, str, int] Truncate all rows before this index value.
- **after** [date, str, int] Truncate all rows after this index value.
- **axis** [[0 or ‘index’, 1 or ‘columns’], optional] Axis to truncate. Truncates the index (rows) by default.
- **copy** [bool, default is True.] Return a copy of the truncated section.

Returns  
- type of caller The truncated Series or DataFrame.

See also:  
- **DataFrame.loc** Select a subset of a DataFrame by label.  
- **DataFrame.iloc** Select a subset of a DataFrame by position.

Notes  
If the index being truncated contains only datetime values, **before** and **after** may be specified as strings instead of Timestamps.

Examples

```python
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
                    ...         'B': ['f', 'g', 'h', 'i', 'j'],
                    ...         'C': ['k', 'l', 'm', 'n', 'o']},
                    ... index=[1, 2, 3, 4, 5])
  A B C
1 a f k
2 b g l
3 c h m
4 d i n
5 e j o

>>> df.truncate(before=2, after=4)
  A B C
2 b g l
3 c h m
4 d i n

The columns of a DataFrame can be truncated.
>>> df.truncate(before="A", after="B", axis="columns")

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>h</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>i</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>j</td>
</tr>
</tbody>
</table>

For Series, only rows can be truncated.

>>> df['A'].truncate(before=2, after=4)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

Name: A, dtype: object

The index values in truncate can be datetimes or string dates.

>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-31 23:59:56</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-31 23:59:57</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-31 23:59:58</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-31 23:59:59</td>
<td>1</td>
</tr>
<tr>
<td>2016-02-01 00:00:00</td>
<td>1</td>
</tr>
</tbody>
</table>

>>> df.truncate(before=pd.Timestamp('2016-01-05'), ...
<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-09 23:59:56</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:57</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:58</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:59</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-10 00:00:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Because the index is a DatetimeIndex containing only dates, we can specify before and after as strings. They will be coerced to Timestamps before truncation.

>>> df.truncate(before='2016-01-05', '2016-01-10').tail()

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-09 23:59:56</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:57</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:58</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-09 23:59:59</td>
<td>1</td>
</tr>
<tr>
<td>2016-01-10 00:00:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that truncate assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

>>> df.loc['2016-01-05':'2016-01-10', :].tail()

(continues on next page)
pandas.Series.tshift

pandas.Series.tshift(periods=1, freq=None, axis=0)

Shift the time index, using the index’s frequency if available.

Deprecated since version 1.1.0: Use shift instead.

Parameters

- **periods** [int] Number of periods to move, can be positive or negative.
- **freq** [DateOffset, timedelta, or str, default None] Increment to use from the tseries module or time rule expressed as a string (e.g. ‘EOM’).
- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default 0] Corresponds to the axis that contains the Index.

Returns

- **shifted** [Series/DataFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

pandas.Series.tz_convert

pandas.Series.tz_convert(tz, axis=0, level=None, copy=True)

Convert tz-aware axis to target time zone.

Parameters

- **tz** [str or tzinfo object]
- **axis** [the axis to convert]
- **level** [int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.
- **copy** [bool, default True] Also make a copy of the underlying data.

Returns

- **{klass}** Object with time zone converted axis.

Raises

- **TypeError** If the axis is tz-naive.
pandas.Series.tz_localize

Series.tz_localize(tz, axis=0, level=None, copy=True, ambiguous='raise', nonexistent='raise')

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use Series.dt.tz_localize().

Parameters

- **tz** [str or tzinfo]
- **axis** [the axis to localize]
- **level** [int, str, default None] If axis is a MultiIndex, localize a specific level. Otherwise must be None.
- **copy** [bool, default True] Also make a copy of the underlying data.
- **ambiguous** ['infer', bool ndarray, 'NaT', default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the ambiguous parameter dictates how ambiguous times should be handled.
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
- **nonexistent** [str, default ‘raise’] A nonexistent time does not exist in a particular time zone where clocks moved forward due to DST. Valid values are:
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return NaT where there are nonexistent times
  - timedelta objects will shift nonexistent times by the timedelta
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns

Series or DataFrame  Same type as the input.

Raises

- TypeError  If the TimeSeries is tz-aware and tz is not None.
Examples

Localize local times:

```python
>>> s = pd.Series([1],
...               index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00 1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```python
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                         ...
...                                         '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00 0
2018-10-28 02:00:00+02:00 1
2018-10-28 02:30:00+02:00 2
2018-10-28 02:00:00+01:00 3
2018-10-28 02:30:00+01:00 4
2018-10-28 03:00:00+01:00 5
2018-10-28 03:30:00+01:00 6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly:

```python
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                         ...
...                                         '2018-10-28 03:46:00']))
>>>
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 03:00:00+02:00 0
2018-10-28 03:30:00+01:00 1
2018-10-28 03:46:00+01:00 2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or ‘shift_forward’ or ‘shift_backward’.

```python
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                         ...
...                                         '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
```

(continues on next page)
pandas.Series.unique

Series.unique()

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

Returns

ndarray or ExtensionArray  The unique values returned as a NumPy array. See Notes.

See also:

unique  Top-level unique method for any 1-d array-like object.

Index.unique  Return Index with unique values from an Index object.

Notes

Returns the unique values as a NumPy array. In case of an extension-array backed Series, a new ExtensionArray of that type with just the unique values is returned. This includes

- Categorical
  - Period
  - Datetime with Timezone
  - Interval
  - Sparse
  - IntegerNA

See Examples section.

Examples

```python
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])

>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
... for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]
```

An Categorical will return categories in the order of appearance and with the same dtype.

```python
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[ 'b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
... ordered=True)).unique()
[ 'b', 'a', 'c']
Categories (3, object): ['a' < 'b' < 'c']
```

### pandas.Series.unstack

Series.unstack(level=-1, fill_value=None)

Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.

- **Parameters**
  - *level* [int, str, or list of these, default last level] Level(s) to unstack, can pass level name.
  - *fill_value* [scalar value, default None] Value to use when replacing NaN values.

- **Returns**
  - **DataFrame** Unstacked Series.

### Notes

Reference *the user guide* for more examples.

### Examples

```python
>>> s = pd.Series([1, 2, 3, 4],
... index=pd.MultiIndex.from_product([['one', 'two'],
... ['a', 'b']]))
```

```python
>>> s

<table>
<thead>
<tr>
<th>one</th>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>two</td>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>4</td>
</tr>
</tbody>
</table>

| dtype: int64

```python
>>> s.unstack(level=-1)
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
```
>>> s.unstack(level=0)
  one  two  
 a   1  3  
 b   2  4  

pandas.Series.update

Series.update(other)
Modify Series in place using values from passed Series.
Uses non-NA values from passed Series to make updates. Aligns on index.

Parameters

other [Series, or object coercible into Series]

Examples

>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6]))
>>> s
0   4
1   5
2   6
dtype: int64

>>> s = pd.Series(['a', 'b', 'c'])
>>> s.update(pd.Series(['d', 'e'], index=[0, 2]))
>>> s
0   d
1   b
2   e
dtype: object

>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6, 7, 8]))
>>> s
0   4
1   5
2   6
dtype: int64

If other contains NaNs the corresponding values are not updated in the original Series.

>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, np.nan, 6]))
>>> s
0   4
1   2
2   6
dtype: int64
other can also be a non-Series object type that is coercible into a Series

```python
>>> s = pd.Series([1, 2, 3])
>>> s.update([4, np.nan, 6])
>>> s
0  4
1  2
2  6
dtype: int64
```

```python
>>> s = pd.Series([1, 2, 3])
>>> s.update({1: 9})
>>> s
0  1
1  9
2  3
dtype: int64
```

**pandas.Series.value_counts**

Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

**Parameters**

- `normalize` [bool, default False] If True then the object returned will contain the relative frequencies of the unique values.
- `sort` [bool, default True] Sort by frequencies.
- `ascending` [bool, default False] Sort in ascending order.
- `bins` [int, optional] Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data.
- `dropna` [bool, default True] Don’t include counts of NaN.

**Returns**

Series

**See also:**

- `Series.count` Number of non-NA elements in a Series.
- `DataFrame.count` Number of non-NA elements in a DataFrame.
- `DataFrame.value_counts` Equivalent method on DataFrames.
Examples

>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0  2
1.0  1
2.0  1
4.0  1
dtype: int64

With normalize set to True, returns the relative frequency by dividing all values by the sum of values.

>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
>>> s.value_counts(normalize=True)
3.0  0.4
1.0  0.2
2.0  0.2
4.0  0.2
dtype: float64

bins

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

>>> s.value_counts(bins=3)
(0.996, 2.0]  2
(2.0, 3.0]  2
(3.0, 4.0]  1
dtype: int64

dropna

With dropna set to False we can also see NaN index values.

>>> s.value_counts(dropna=False)
3.0  2
1.0  1
2.0  1
4.0  1
NaN  1
dtype: int64

pandas.Series.var

Series.var(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis  [[index (0)]]

skipna  [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

ddf [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

scalar or Series (if level specified)

Examples

```python
def = pd.DataFrame({'person_id': [0, 1, 2, 3],
                   'age': [21, 25, 62, 43],
                   'height': [1.61, 1.87, 1.49, 2.01]})
def.set_index('person_id')
```

```
  age  height
person_id
0   21    1.61
1   25    1.87
2   62    1.49
3   43    2.01
```

```python
def.var()
```

```
age   352.916667
height 0.056367
```

Alternatively, \( ddof=0 \) can be set to normalize by \( N \) instead of \( N-1 \):

```python
def.var(ddof=0)
```

```
age   264.687500
height 0.042275
```

pandas.Series.view

Series.view(dtype=None)

Create a new view of the Series.

This function will return a new Series with a view of the same underlying values in memory, optionally reinterpreted with a new data type. The new data type must preserve the same size in bytes as to not cause index misalignment.

Parameters

- **dtype** [data type] Data type object or one of their string representations.

Returns

- **Series** A new Series object as a view of the same data in memory.

See also:
**numpy.ndarray.view** Equivalent numpy function to create a new view of the same data in memory.

**Notes**

Series are instantiated with `dtype=float64` by default. While `numpy.ndarray.view()` will return a view with the same data type as the original array, `Series.view()` (without specified `dtype`) will try using `float64` and may fail if the original data type size in bytes is not the same.

**Examples**

```python
>>> s = pd.Series([-2, -1, 0, 1, 2], dtype='int8')
>>> s
0   -2
1   -1
2    0
3    1
4    2
dtype: int8
```

The 8 bit signed integer representation of -1 is `0b11111111`, but the same bytes represent 255 if read as an 8 bit unsigned integer:

```python
>>> us = s.view('uint8')
>>> us
0   254
1   255
2    0
3    1
4    2
dtype: uint8
```

The views share the same underlying values:

```python
>>> us[0] = 128
>>> s
0   -128
1    -1
2     0
3    1
4    2
dtype: int8
```
pandas.Series.where

Series.where(cond=NoDefault.no_default, other=None, inplace=False, axis=None, errors='raise', try_cast=None)

Replace values where the condition is False.

Parameters

- **cond** [bool Series/DataFrame, array-like, or callable] Where cond is True, keep the original value. Where False, replace with corresponding value from other. If cond is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

- **other** [scalar, Series/DataFrame, or callable] Entries where cond is False are replaced with corresponding value from other. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

- **inplace** [bool, default False] Whether to perform the operation in place on the data.

- **axis** [int, default None] Alignment axis if needed.

- **level** [int, default None] Alignment level if needed.

- **errors** [str, {'raise', 'ignore'}, default 'raise'] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.
  - ‘raise’ : allow exceptions to be raised.
  - ‘ignore’ : suppress exceptions. On error return original object.

- **try_cast** [bool, default None] Try to cast the result back to the input type (if possible).
  Deprecated since version 1.3.0: Manually cast back if necessary.

Returns

Same type as caller or None if inplace=True.

See also:

- **DataFrame.mask()** Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if cond is True the element is used; otherwise the corresponding element from the DataFrame other is used.

The signature for DataFrame.where() differs from numpy.where(). Roughly df1.where(m, df2) is equivalent to np.where(m, df1, df2).

For further details and examples see the where documentation in indexing.
Examples

```python
t s = pd.Series(range(5))
>>> s.where(s > 0)
0   NaN
1   1.0
2   2.0
3   3.0
4   4.0
dtype: float64
>>> s.mask(s > 0)
0   0.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
>>> s.where(s > 1, 10)
0  10
1  10
2   2
3   3
4   4
dtype: int64
>>> s.mask(s > 1, 10)
0   0
1   1
2  10
3  10
4  10
dtype: int64
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0   0 -1
1 -2  3
2  4 -5
3  6 -7
4  8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A   B
0  True  True
```

(continues on next page)
1 True True
2 True True
3 True True
4 True True

```python
df.where(m, -df) == df.mask(~m, -df)
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

### pandas.Series.xs

**Series.xs(key, axis=0, level=None, drop_level=True)**

Return cross-section from the Series/DataFrame.

This method takes a `key` argument to select data at a particular level of a MultiIndex.

**Parameters**

- **key** [label or tuple of label] Label contained in the index, or partially in a MultiIndex.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Axis to retrieve cross-section on.
- **level** [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.
- **drop_level** [bool, default True] If False, returns object with same levels as self.

**Returns**

**Series or DataFrame** Cross-section from the original Series or DataFrame corresponding to the selected index levels.

**See also:**

- **DataFrame.loc** Access a group of rows and columns by label(s) or a boolean array.
- **DataFrame.iloc** Purely integer-location based indexing for selection by position.

**Notes**

.xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of .xs functionality, see *MultiIndex Slicers.*
Examples

```python
>>> d = {'num_legs': [4, 4, 2, 2],  
...     'num_wings': [0, 0, 2, 2],  
...     'class': ['mammal', 'mammal', 'mammal', 'bird'],  
...     'animal': ['cat', 'dog', 'bat', 'penguin'],  
...     'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df

<table>
<thead>
<tr>
<th>class</th>
<th>animal</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>mammal</td>
<td>cat</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dog</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bat</td>
<td>flies</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>bird</td>
<td>penguin</td>
<td>walks</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Get values at specified index

```python
>>> df.xs('mammal')

<table>
<thead>
<tr>
<th>animal</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>dog</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>bat</td>
<td>flies</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Get values at several indexes

```python
>>> df.xs(('mammal', 'dog'))

<table>
<thead>
<tr>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Get values at specified index and level

```python
>>> df.xs('cat', level=1)

<table>
<thead>
<tr>
<th>class</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>mammal</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Get values at several indexes and levels

```python
>>> df.xs(('bird', 'walks'),  
...     level=[0, 'locomotion'])

<table>
<thead>
<tr>
<th>animal</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>penguin</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Get values at specified column and axis

```python
>>> df.xs('num_wings', axis=1)

<table>
<thead>
<tr>
<th>class</th>
<th>animal</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>mammal</td>
<td>cat</td>
<td>walks</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dog</td>
<td>walks</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
bat  flies  2
bird  penguin  walks  2
Name: num_wings, dtype: int64

### 3.3.2 Attributes

#### Axes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.index</code></td>
<td>The index (axis labels) of the Series.</td>
</tr>
<tr>
<td><code>Series.array</code></td>
<td>The ExtensionArray of the data backing this Series or Index.</td>
</tr>
<tr>
<td><code>Series.values</code></td>
<td>Return Series as ndarray or ndarray-like depending on the dtype.</td>
</tr>
<tr>
<td><code>Series.dtype</code></td>
<td>Return the dtype object of the underlying data.</td>
</tr>
<tr>
<td><code>Series.shape</code></td>
<td>Return a tuple of the shape of the underlying data.</td>
</tr>
<tr>
<td><code>Series nbytes</code></td>
<td>Return the number of bytes in the underlying data.</td>
</tr>
<tr>
<td><code>Series.ndim</code></td>
<td>Number of dimensions of the underlying data, by definition 1.</td>
</tr>
<tr>
<td><code>Series.size</code></td>
<td>Return the number of elements in the underlying data.</td>
</tr>
<tr>
<td><code>Series.T</code></td>
<td>Return the transpose, which is by definition self.</td>
</tr>
<tr>
<td><code>Series.memory_usage(index, deep)</code></td>
<td>Return the memory usage of the Series.</td>
</tr>
<tr>
<td><code>Series.hasnans</code></td>
<td>Return True if there are any NaNs.</td>
</tr>
<tr>
<td><code>Series.dtypes</code></td>
<td>Return the dtype object of the underlying data.</td>
</tr>
<tr>
<td><code>Series.name</code></td>
<td>Return the name of the Series.</td>
</tr>
<tr>
<td><code>Series.flags</code></td>
<td>Get the properties associated with this pandas object.</td>
</tr>
<tr>
<td><code>Series.set_flags(*[, copy, ...])</code></td>
<td>Return a new object with updated flags.</td>
</tr>
</tbody>
</table>

**pandas.Series.empty**

**property** `Series.empty`

Indicator whether Series/DataFrame is empty.

- True if Series/DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

**Returns**

- `bool` If Series/DataFrame is empty, return True, if not return False.

**See also:**

- `Series.dropna` Return series without null values.
- `DataFrame.dropna` Return DataFrame with labels on given axis omitted where (all or any) data are missing.
Notes

If Series/DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```python
>>> df_empty = pd.DataFrame({'A': []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```python
>>> df = pd.DataFrame({'A': [np.nan]})
>>> df
   A
0  NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

```python
>>> ser_empty = pd.Series({'A': []})
>>> ser_empty
A []
dtype: object
>>> ser_empty.empty
False
>>> ser_empty = pd.Series()
>>> ser_empty.empty
True
```
3.3.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.astype(dtype[, copy, errors])</code></td>
<td>Cast a pandas object to a specified dtype <code>dtype</code>.</td>
</tr>
<tr>
<td><code>Series.convert_dtypes([infer_objects, ...])</code></td>
<td>Convert columns to best possible dtypes using dtypes supporting <code>pd.NA</code>.</td>
</tr>
<tr>
<td><code>Series.infer_objects()</code></td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td><code>Series.copy([deep])</code></td>
<td>Make a copy of this object's indices and data.</td>
</tr>
<tr>
<td><code>Series.bool()</code></td>
<td>Return the bool of a single element Series or DataFrame.</td>
</tr>
<tr>
<td><code>Series.to_numpy([dtype, copy, na_value])</code></td>
<td>A NumPy ndarray representing the values in this Series or Index.</td>
</tr>
<tr>
<td><code>Series.to_period([freq, copy])</code></td>
<td>Convert Series from DatetimeIndex to PeriodIndex.</td>
</tr>
<tr>
<td><code>Series.to_timestamp([freq, how, copy])</code></td>
<td>Cast to DatetimeIndex of Timestamps, at <code>beginning</code> of period.</td>
</tr>
<tr>
<td><code>Series.to_list()</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>Series.__array__((dtype))</code></td>
<td>Return the values as a NumPy array.</td>
</tr>
</tbody>
</table>

### pandas.Series.__array__

**Series.__array__(dtype=None)**

Return the values as a NumPy array.

Users should not call this directly. Rather, it is invoked by `numpy.array()` and `numpy.asarray()`.

**Parameters**

- **dtype** [str or numpy.dtype, optional] The dtype to use for the resulting NumPy array. By default, the dtype is inferred from the data.

**Returns**

- **numpy.ndarray** The values in the series converted to a `numpy.ndarray` with the specified `dtype`.

**See also:**

- `array` Create a new array from data.
- `Series.array` Zero-copy view to the array backing the Series.
- `Series.to_numpy` Series method for similar behavior.

**Examples**

```python
>>> ser = pd.Series([1, 2, 3])
>>> np.asarray(ser)
array([1, 2, 3])
```

For timezone-aware data, the timezones may be retained with `dtype='object'`

```python
>>> tzser = pd.Series(pd.date_range('2000', periods=2, tz='CET'))
>>> np.asarray(tzser, dtype='object')
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET')],
      dtype=object)
```

Or the values may be localized to UTC and the tzinfo discarded with `dtype='datetime64[ns]'`
3.3.4 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.get()</code></td>
<td>Get item from object for given key (ex: DataFrame column).</td>
</tr>
<tr>
<td><code>Series.at</code></td>
<td>Access a single value for a row/column label pair.</td>
</tr>
<tr>
<td><code>Series.iat</code></td>
<td>Access a single value for a row/column pair by integer position.</td>
</tr>
<tr>
<td><code>Series.loc</code></td>
<td>Access a group of rows and columns by label(s) or a boolean array.</td>
</tr>
<tr>
<td><code>Series.iloc</code></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><code>Series.__iter__()</code></td>
<td>Return an iterator of the values.</td>
</tr>
<tr>
<td><code>Series.items()</code></td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td><code>Series.iteritems()</code></td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td><code>Series.keys()</code></td>
<td>Return alias for index.</td>
</tr>
<tr>
<td><code>Series.pop()</code></td>
<td>Return item and drops from series.</td>
</tr>
<tr>
<td><code>Series.item()</code></td>
<td>Return the first element of the underlying data as a Python scalar.</td>
</tr>
<tr>
<td><code>Series.xs()</code></td>
<td>Return cross-section from the Series/DataFrame.</td>
</tr>
</tbody>
</table>

**pandas.Series.__iter__**

`Series.__iter__()`  

Return an iterator of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

**Returns**

- `iterator`

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the *indexing documentation*. 
### 3.3.5 Binary operator functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.add</code></td>
<td>Return Addition of series and other, element-wise (binary operator <code>add</code>).</td>
</tr>
<tr>
<td><code>Series.sub</code></td>
<td>Return Subtraction of series and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>Series.mul</code></td>
<td>Return Multiplication of series and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>Series.div</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>Series.truediv</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>Series.floordiv</code></td>
<td>Return Integer division of series and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>Series.mod</code></td>
<td>Return Modulo of series and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>Series.pow</code></td>
<td>Return Exponential power of series and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>Series.radd</code></td>
<td>Return Addition of series and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>Series.rsub</code></td>
<td>Return Subtraction of series and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>Series.rmul</code></td>
<td>Return Multiplication of series and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>Series.rdiv</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>rtrue_div</code>).</td>
</tr>
<tr>
<td><code>Series.rtruediv</code></td>
<td>Return Floating division of series and other, element-wise (binary operator <code>rtrue_div</code>).</td>
</tr>
<tr>
<td><code>Series.rfloordiv</code></td>
<td>Return Integer division of series and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>Series.rmod</code></td>
<td>Return Modulo of series and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>Series.rpow</code></td>
<td>Return Exponential power of series and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>Series.combine</code></td>
<td>Combine the Series with a Series or scalar according to <code>func</code>.</td>
</tr>
<tr>
<td><code>Series.combine_first</code></td>
<td>Update null elements with value in the same location in 'other'.</td>
</tr>
<tr>
<td><code>Series.round</code></td>
<td>Round each value in a Series to the given number of decimals.</td>
</tr>
<tr>
<td><code>Series.lt</code></td>
<td>Return Less than of series and other, element-wise (binary operator <code>lt</code>).</td>
</tr>
<tr>
<td><code>Series.gt</code></td>
<td>Return Greater than of series and other, element-wise (binary operator <code>gt</code>).</td>
</tr>
<tr>
<td><code>Series.le</code></td>
<td>Return Less than or equal to of series and other, element-wise (binary operator <code>le</code>).</td>
</tr>
<tr>
<td><code>Series.ge</code></td>
<td>Return Greater than or equal to of series and other, element-wise (binary operator <code>ge</code>).</td>
</tr>
<tr>
<td><code>Series.ne</code></td>
<td>Return Not equal to of series and other, element-wise (binary operator <code>ne</code>).</td>
</tr>
<tr>
<td><code>Series.eq</code></td>
<td>Return Equal to of series and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
<tr>
<td><code>Series.product</code></td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.dot</code></td>
<td>Compute the dot product between the Series and the columns of other.</td>
</tr>
</tbody>
</table>
### 3.3.6 Function application, GroupBy & window

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.apply</strong></td>
<td>Invoke function on values of Series.</td>
</tr>
<tr>
<td><strong>Series.agg</strong></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><strong>Series.aggregate</strong></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><strong>Series.transform</strong></td>
<td>Call func on self producing a Series with the same axis shape as self.</td>
</tr>
<tr>
<td><strong>Series.map</strong></td>
<td>Map values of Series according to an input mapping or function.</td>
</tr>
<tr>
<td><strong>Series.groupby</strong></td>
<td>Group Series using a mapper or by a Series of columns.</td>
</tr>
<tr>
<td><strong>Series.rolling</strong></td>
<td>Provide rolling window calculations.</td>
</tr>
<tr>
<td><strong>Series.expanding</strong></td>
<td>Provide expanding window calculations.</td>
</tr>
<tr>
<td><strong>Series.ewm</strong></td>
<td>Provide exponentially weighted (EW) calculations.</td>
</tr>
<tr>
<td><strong>Series.pipe</strong></td>
<td>Apply chainable functions that expect Series or DataFrames.</td>
</tr>
</tbody>
</table>

### 3.3.7 Computations / descriptive stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.abs()</strong></td>
<td>Return a Series/DataFrame with absolute numeric value of each element.</td>
</tr>
<tr>
<td><strong>Series.all</strong></td>
<td>Return whether all elements are True, potentially over an axis.</td>
</tr>
<tr>
<td><strong>Series.any</strong></td>
<td>Return whether any element is True, potentially over an axis.</td>
</tr>
<tr>
<td><strong>Series.autocorr</strong></td>
<td>Compute the lag-N autocorrelation.</td>
</tr>
<tr>
<td><strong>Series.between</strong></td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right.</td>
</tr>
<tr>
<td><strong>Series.clip</strong></td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><strong>Series.corr</strong></td>
<td>Compute correlation with other Series, excluding missing values.</td>
</tr>
<tr>
<td><strong>Series.count</strong></td>
<td>Return number of non-NA/null observations in the Series.</td>
</tr>
<tr>
<td><strong>Series.cov</strong></td>
<td>Compute covariance with Series, excluding missing values.</td>
</tr>
<tr>
<td><strong>Series.cummax</strong></td>
<td>Return cumulative maximum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cummin</strong></td>
<td>Return cumulative minimum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cumprod</strong></td>
<td>Return cumulative product over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cumsum</strong></td>
<td>Return cumulative sum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.describe</strong></td>
<td>Generate descriptive statistics.</td>
</tr>
<tr>
<td><strong>Series.diff</strong></td>
<td>First discrete difference of element.</td>
</tr>
<tr>
<td><strong>Series.factorize</strong></td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><strong>Series.kurt</strong></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
</tbody>
</table>

continues on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.mad</code></td>
<td>Return the mean absolute deviation of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.max</code></td>
<td>Return the maximum of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.mean</code></td>
<td>Return the mean of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.median</code></td>
<td>Return the median of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.min</code></td>
<td>Return the minimum of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.mode</code></td>
<td>Return the mode(s) of the Series.</td>
</tr>
<tr>
<td><code>Series.nlargest</code></td>
<td>Return the largest n elements.</td>
</tr>
<tr>
<td><code>Series.nsmallest</code></td>
<td>Return the smallest n elements.</td>
</tr>
<tr>
<td><code>Series.pct_change</code></td>
<td>Percentage change between the current and a prior element.</td>
</tr>
<tr>
<td><code>Series.prod</code></td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.quantile</code></td>
<td>Return value at the given quantile.</td>
</tr>
<tr>
<td><code>Series.rank</code></td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td><code>Series.sem</code></td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><code>Series.skew</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>Series.std</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>Series.sum</code></td>
<td>Return the sum of the values over the requested axis.</td>
</tr>
<tr>
<td><code>Series.var</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>Series.kurtosis</code></td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td><code>Series.unique()</code></td>
<td>Return unique values of Series object.</td>
</tr>
<tr>
<td><code>Series.nunique</code></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><code>Series.is_unique</code></td>
<td>Return boolean if values in the object are unique.</td>
</tr>
<tr>
<td><code>Series.is_monotonic</code></td>
<td>Return boolean if values in the object are monotonic_increasing.</td>
</tr>
<tr>
<td><code>Series.is_monotonic_increasing</code></td>
<td>Alias for is_monotonic.</td>
</tr>
<tr>
<td><code>Series.is_monotonic_decreasing</code></td>
<td>Return boolean if values in the object are monotonic_decreasing.</td>
</tr>
<tr>
<td><code>Series.value_counts</code></td>
<td>Return a Series containing counts of unique values.</td>
</tr>
</tbody>
</table>
### 3.3.8 Reindexing / selection / label manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.align</code></td>
<td>Align two objects on their axes with the specified join method.</td>
</tr>
<tr>
<td><code>Series.drop</code></td>
<td>Return Series with specified index labels removed.</td>
</tr>
<tr>
<td><code>Series.drop_duplicates</code></td>
<td>Return Series/DataFrame with requested index / column level(s) removed.</td>
</tr>
<tr>
<td><code>Series.duplicated</code></td>
<td>Return Series with duplicate values removed.</td>
</tr>
<tr>
<td><code>Series.equals</code></td>
<td>Test whether two objects contain the same elements.</td>
</tr>
<tr>
<td><code>Series.first</code></td>
<td>Select initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>Series.head</code></td>
<td>Return the first ( n ) rows.</td>
</tr>
<tr>
<td><code>Series.idxmax</code></td>
<td>Return the row label of the maximum value.</td>
</tr>
<tr>
<td><code>Series.idxmin</code></td>
<td>Return the row label of the minimum value.</td>
</tr>
<tr>
<td><code>Series.isin</code></td>
<td>Whether elements in Series are contained in ( \text{values} ).</td>
</tr>
<tr>
<td><code>Series.last</code></td>
<td>Select final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>Series.reindex</code></td>
<td>Conform Series to new index with optional filling logic.</td>
</tr>
<tr>
<td><code>Series.reindex_like</code></td>
<td>Return an object with matching indices as other object.</td>
</tr>
<tr>
<td><code>Series.rename</code></td>
<td>Alter Series index labels or name.</td>
</tr>
<tr>
<td><code>Series.rename_axis</code></td>
<td>Set the name of the axis for the index or columns.</td>
</tr>
<tr>
<td><code>Series.reset_index</code></td>
<td>Generate a new DataFrame or Series with the index reset.</td>
</tr>
<tr>
<td><code>Series.sample</code></td>
<td>Return a random sample of items from an axis of object.</td>
</tr>
<tr>
<td><code>Series.set_axis</code></td>
<td>Assign desired index to given axis.</td>
</tr>
<tr>
<td><code>Series.take</code></td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><code>Series.tail</code></td>
<td>Return the last ( n ) rows.</td>
</tr>
<tr>
<td><code>Series.truncate</code></td>
<td>Truncate a Series or DataFrame before and after some index value.</td>
</tr>
<tr>
<td><code>Series.where</code></td>
<td>Replace values where the condition is False.</td>
</tr>
<tr>
<td><code>Series.mask</code></td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td><code>Series.add_prefix</code></td>
<td>Prefix labels with string ( \text{prefix} ).</td>
</tr>
<tr>
<td><code>Series.add_suffix</code></td>
<td>Suffix labels with string ( \text{suffix} ).</td>
</tr>
<tr>
<td><code>Series.filter</code></td>
<td>Subset the dataframe rows or columns according to the specified index labels.</td>
</tr>
</tbody>
</table>
3.3.9 Missing data handling

- **Series.backfill**([axis, inplace, limit, downcast]) Synonym for **DataFrame.fillna()** with method='bfill'.
- **Series.bfill**([axis, inplace, limit, downcast]) Synonym for **DataFrame.fillna()** with method='bfill'.
- **Series.dropna**([axis, inplace, how]) Return a new Series with missing values removed.
- **Series.ffill**([axis, inplace, limit, downcast]) Synonym for **DataFrame.fillna()** with method='ffill'.
- **Series.fillna**([value, method, axis, ...]) Fill NA/Nan values using the specified method.
- **Series.interpolate**([method, axis, limit, ...]) Fill NaN values using an interpolation method.
- **Series.isna**() Detect missing values.
- **Series.isnull**() Series.isnull is an alias for Series.isna.
- **Series.notna**() Detect existing (non-missing) values.
- **Series.notnull**() Series.notnull is an alias for Series.notna.
- **Series.pad**([axis, inplace, limit, downcast]) Synonym for **DataFrame.fillna()** with method='ffill'.
- **Series.replace**([to_replace, value, inplace, ...]) Replace values given in to_replace with value.

3.3.10 Reshaping, sorting

- **Series.argsort**([axis, kind, order]) Return the integer indices that would sort the Series values.
- **Series.argmin**([axis, skipna]) Return int position of the smallest value in the Series.
- **Series.argmax**([axis, skipna]) Return int position of the largest value in the Series.
- **Series.reorder_levels**(order) Rearrange index levels using input order.
- **Series.sort_values**([axis, ascending, ...]) Sort by the values.
- **Series.sort_index**([axis, level, ascending, ...]) Sort Series by index labels.
- **Series.swaplevel**(i, j, copy) Swap levels i and j in a MultiIndex.
- **Series.unstack**(level, fill_value) Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.
- **Series.explode**(ignore_index) Transform each element of a list-like to a row.
- **Series.searchsorted**(value[, side, sorter]) Find indices where elements should be inserted to maintain order.
- **Series.ravel**(order) Return the flattened underlying data as an ndarray.
- **Series.repeat**(repeats[, axis]) Repeat elements of a Series.
- **Series.squeeze**(axis) Squeeze 1 dimensional axis objects into scalars.
- **Series.view**(dtype) Create a new view of the Series.

3.3.11 Combining / comparing / joining / merging

- **Series.append**([to_append[, ignore_index, ...]]) (DEPRECATED) Concatenate two or more Series.
- **Series.compare**(other[, align_axis, ...]) Compare to another Series and show the differences.
- **Series.update**(other) Modify Series in place using values from passed Series.
### 3.3.12 Time Series-related

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.asfreq(freq[, method, how, ...])</td>
<td>Convert time series to specified frequency.</td>
</tr>
<tr>
<td>Series.asof(where[, subset])</td>
<td>Return the last row(s) without any NaNs before where.</td>
</tr>
<tr>
<td>Series.shift([periods, freq, axis, fill_value])</td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
<tr>
<td>Series.first_valid_index()</td>
<td>Return index for first non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td>Series.last_valid_index()</td>
<td>Return index for last non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td>Series.resample(rule[, axis, closed, label, ...])</td>
<td>Resample time-series data.</td>
</tr>
<tr>
<td>Series.tz_convert(tz[, axis, level, copy])</td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td>Series.tz_localize(tz[, axis, level, copy, ...])</td>
<td>Localize tz-naive index of a Series or DataFrame to target time zone.</td>
</tr>
<tr>
<td>Series.at_time(time[, asof, axis])</td>
<td>Select values at particular time of day (e.g., 9:30AM).</td>
</tr>
<tr>
<td>Series.between_time(start_time, end_time[, ...])</td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
</tr>
<tr>
<td>Series.tshift([periods, freq, axis])</td>
<td>(DEPRECATED) Shift the time index, using the index’s frequency if available.</td>
</tr>
<tr>
<td>Series.slice_shift([periods, axis])</td>
<td>(DEPRECATED) Equivalent to shift without copying data.</td>
</tr>
</tbody>
</table>

### 3.3.13 Accessors

Pandas provides dtype-specific methods under various accessors. These are separate namespaces within Series that only apply to specific data types.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Accessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime, Timedelta, Period</td>
<td>dt</td>
</tr>
<tr>
<td>String</td>
<td>str</td>
</tr>
<tr>
<td>Categorical</td>
<td>cat</td>
</tr>
<tr>
<td>Sparse</td>
<td>sparse</td>
</tr>
</tbody>
</table>

**Datetimelike properties**

Series.dt can be used to access the values of the series as datetimelike and return several properties. These can be accessed like Series.dt.<property>.
## Datetime properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.dt.date</code></td>
<td>Returns numpy array of python <code>datetime.date</code> objects.</td>
</tr>
<tr>
<td><code>Series.dt.time</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects.</td>
</tr>
<tr>
<td><code>Series.dt.timetz</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects with timezone information.</td>
</tr>
<tr>
<td><code>Series.dt.year</code></td>
<td>The year of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.month</code></td>
<td>The month as January=1, December=12.</td>
</tr>
<tr>
<td><code>Series.dt.day</code></td>
<td>The day of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.hour</code></td>
<td>The hours of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.minute</code></td>
<td>The minutes of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.second</code></td>
<td>The seconds of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.microsecond</code></td>
<td>The microseconds of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.nanosecond</code></td>
<td>The nanoseconds of the datetime.</td>
</tr>
<tr>
<td><code>Series.dt.week</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>Series.dt.weekofyear</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>Series.dt.dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>Series.dt.day_of_week</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>Series.dt.weekday</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>Series.dt.day_of_year</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>Series.dt.quarter</code></td>
<td>The quarter of the date.</td>
</tr>
<tr>
<td><code>Series.dt.is_month_start</code></td>
<td>Indicates whether the date is the first day of the month.</td>
</tr>
<tr>
<td><code>Series.dt.is_month_end</code></td>
<td>Indicates whether the date is the last day of the month.</td>
</tr>
<tr>
<td><code>Series.dt.is_quarter_start</code></td>
<td>Indicator for whether the date is the first day of a quarter.</td>
</tr>
<tr>
<td><code>Series.dt.is_quarter_end</code></td>
<td>Indicator for whether the date is the last day of a quarter.</td>
</tr>
<tr>
<td><code>Series.dt.is_year_start</code></td>
<td>Indicate whether the date is the first day of a year.</td>
</tr>
<tr>
<td><code>Series.dt.is_year_end</code></td>
<td>Indicate whether the date is the last day of the year.</td>
</tr>
<tr>
<td><code>Series.dt.is_leap_year</code></td>
<td>Boolean indicator if the date belongs to a leap year.</td>
</tr>
<tr>
<td><code>Series.dt.daysinmonth</code></td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td><code>Series.dt.days_in_month</code></td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td><code>Series.dt.tz</code></td>
<td>Return the timezone.</td>
</tr>
<tr>
<td><code>Series.dt.freq</code></td>
<td>Return the frequency object for this PeriodArray.</td>
</tr>
</tbody>
</table>

### pandas.Series.dt.date

`Series.dt.date` returns numpy array of python `datetime.date` objects.

- Namely, the date part of Timestamps without time and timezone information.
**pandas.Series.dt.time**

`Series.dt.time`

Returns numpy array of `datetime.time` objects.

The time part of the Timestamps.

**pandas.Series.dt.timetz**

`Series.dt.timetz`

Returns numpy array of `datetime.time` objects with timezone information.

The time part of the Timestamps.

**pandas.Series.dt.year**

`Series.dt.year`

The year of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
...   pd.date_range("2000-01-01", periods=3, freq="Y")
... )
>>> datetime_series
0 2000-12-31
1 2001-12-31
2 2002-12-31
dtype: datetime64[ns]
>>> datetime_series.dt.year
0 2000
1 2001
2 2002
dtype: int64
```

**pandas.Series.dt.month**

`Series.dt.month`

The month as January=1, December=12.
Examples

```python
>>> datetime_series = pd.Series(
    ...     pd.date_range("2000-01-01", periods=3, freq="M")
    ... )
>>> datetime_series
0   2000-01-31
1   2000-02-29
2   2000-03-31
dtype: datetime64[ns]
>>> datetime_series.dt.month
0    1
1    2
2    3
dtype: int64
```

**pandas.Series.dt.day**

Series.dt.day

The day of the datetime.

Examples

```python
>>> datetime_series = pd.Series(
    ...     pd.date_range("2000-01-01", periods=3, freq="D")
    ... )
>>> datetime_series
0   2000-01-01
1   2000-01-02
2   2000-01-03
dtype: datetime64[ns]
>>> datetime_series.dt.day
0    1
1    2
2    3
dtype: int64
```

**pandas.Series.dt.hour**

Series.dt.hour

The hours of the datetime.
Examples

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="h")
... )
>>> datetime_series
0 2000-01-01 00:00:00
1 2000-01-01 01:00:00
2 2000-01-01 02:00:00
dtype: datetime64[ns]
>>> datetime_series.dt.hour
0 0
1 1
2 2
dtype: int64
```

**pandas.Series.dt.minute**

Series.dt.minute
The minutes of the datetime.

Examples

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="T")
... )
>>> datetime_series
0 2000-01-01 00:00:00
1 2000-01-01 00:01:00
2 2000-01-01 00:02:00
dtype: datetime64[ns]
>>> datetime_series.dt.minute
0 0
1 1
2 2
dtype: int64
```

**pandas.Series.dt.second**

Series.dt.second
The seconds of the datetime.
Examples

```python
>>> datetime_series = pd.Series(
...    pd.date_range("2000-01-01", periods=3, freq="s")
... )
>>> datetime_series
0  2000-01-01 00:00:00
1  2000-01-01 00:00:01
2  2000-01-01 00:00:02
```
dtype: datetime64[ns]
```
>>> datetime_series.dt.second
0    0
1    1
2    2
dtype: int64
```

**pandas.Series.dt.microsecond**

Series.dt.microsecond

The microseconds of the datetime.

Examples

```python
>>> datetime_series = pd.Series(
...    pd.date_range("2000-01-01", periods=3, freq="us")
... )
>>> datetime_series
0  2000-01-01 00:00:00.000000
1  2000-01-01 00:00:00.000001
2  2000-01-01 00:00:00.000002
```
dtype: datetime64[ns]
```
>>> datetime_series.dt.microsecond
0    0
1    1
2    2
dtype: int64
```

**pandas.Series.dt.nanosecond**

Series.dt.nanosecond

The nanoseconds of the datetime.
Examples

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="ns")
... )
>>> datetime_series
0  2000-01-01 00:00:00.000000000
1  2000-01-01 00:00:00.000000001
2  2000-01-01 00:00:00.000000002
dtype: datetime64[ns]
```

```python
>>> datetime_series.dt.nanosecond
0    0
1    1
2    2
dtype: int64
```

**pandas.Series.dt.week**

Series.dt.week

The week ordinal of the year.

Deprecated since version 1.1.0.

Series.dt.weekofyear and Series.dt.week have been deprecated. Please use Series.dt.isocalendar().week instead.

**pandas.Series.dt.weekofyear**

Series.dt.weekofyear

The week ordinal of the year.

Deprecated since version 1.1.0.

Series.dt.weekofyear and Series.dt.week have been deprecated. Please use Series.dt.isocalendar().week instead.

**pandas.Series.dt.dayofweek**

Series.dt.dayofweek

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the `dt` accessor) or DatetimeIndex.

Returns

Series or Index  Containing integers indicating the day number.

See also:

- `Series.dt.dayofweek`  Alias.
- `Series.dt.weekday`  Alias.
- `Series.dt.day_name`  Returns the name of the day of the week.
Examples

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31  5
2017-01-01  6
2017-01-02  0
2017-01-03  1
2017-01-04  2
2017-01-05  3
2017-01-06  4
2017-01-07  5
2017-01-08  6
Freq: D, dtype: int64
```

`pandas.Series.dt.day_of_week`

Series.dt.day_of_week

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the dt accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

- `Series.dt.dayofweek` Alias.
- `Series.dt.weekday` Alias.
- `Series.dt.day_name` Returns the name of the day of the week.

Examples

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31  5
2017-01-01  6
2017-01-02  0
2017-01-03  1
2017-01-04  2
2017-01-05  3
2017-01-06  4
2017-01-07  5
2017-01-08  6
Freq: D, dtype: int64
```
pandas.Series.dt.weekday

Series.dt.weekday
The day of the week with Monday=0, Sunday=6.
Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the dt accessor) or DatetimeIndex.

Returns
Series or Index  Containing integers indicating the day number.

See also:
Series.dt.dayofweek  Alias.
Series.dt.weekday  Alias.
Series.dt.day_name  Returns the name of the day of the week.

Examples

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.Series.dt.dayofyear

Series.dt.dayofyear
The ordinal day of the year.

pandas.Series.dt.day_of_year

Series.dt.day_of_year
The ordinal day of the year.
pandas.Series.dt.quarter

Series.dt.quarter
The quarter of the date.

pandas.Series.dt.is_month_start

Series.dt.is_month_start
Indicates whether the date is the first day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.
is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
dt= 2018-02-27
1 2018-02-28
2 2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0 False
1 False
2 True
dtype: bool
>>> s.dt.is_month_end
0 False
1 True
2 False
dtype: bool
```

```python
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```
pandas.Series.dt.is_month_end

**Series.dt.is_month_end**

Indicates whether the date is the last day of the month.

**Returns**

*Series or array*  For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

**See also:**

*is_month_start*  Return a boolean indicating whether the date is the first day of the month.

*is_month_end*  Return a boolean indicating whether the date is the last day of the month.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0  2018-02-27
1  2018-02-28
2  2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0   False
1   False
2    True
dtype: bool
>>> s.dt.is_month_end
0   False
1    True
2   False
dtype: bool
```

```python
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx
2018-02-27
2018-02-28
2018-03-01
dtype: date
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.Series.dt.is_quarter_start

**Series.dt.is_quarter_start**

Indicator for whether the date is the first day of a quarter.

**Returns**

*is_quarter_start*  [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

*quarter*  Return the quarter of the date.

*is_quarter_end*  Similar property for indicating the quarter start.
Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
df = pd.DataFrame({'dates': pd.date_range('2017-03-30', periods=4)})
df.assign(quarter=df.dates.dt.quarter,
           is_quarter_start=df.dates.dt.is_quarter_start)
```

<table>
<thead>
<tr>
<th>dates</th>
<th>quarter</th>
<th>is_quarter_start</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-03-30</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2017-03-31</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2017-04-01</td>
<td>2</td>
<td>True</td>
</tr>
<tr>
<td>2017-04-02</td>
<td>2</td>
<td>False</td>
</tr>
</tbody>
</table>

```python
idx = pd.date_range('2017-03-30', periods=4)
idx.is_quarter_start
```

```
array([False, False, True, False])
```

**pandas.Series.dt.is_quarter_end**

**Series.dt.is_quarter_end**

Indicator for whether the date is the last day of a quarter.

**Returns**

- **is_quarter_end** [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

- **quarter** Return the quarter of the date.
- **is_quarter_start** Similar property indicating the quarter start.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
df = pd.DataFrame({'dates': pd.date_range('2017-03-30', periods=4)})
df.assign(quarter=df.dates.dt.quarter,
           is_quarter_end=df.dates.dt.is_quarter_end)
```

<table>
<thead>
<tr>
<th>dates</th>
<th>quarter</th>
<th>is_quarter_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-03-30</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2017-03-31</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>2017-04-01</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>2017-04-02</td>
<td>2</td>
<td>False</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')

>>> idx.is_quarter_end
array([False, True, False, False])
```

**pandas.Series.dt.is_year_start**

Series.dt.is_year_start

Indicate whether the date is the first day of a year.

**Returns**

Series or DatetimeIndex  The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

*is_year_end*  Similar property indicating the last day of the year.

**Examples**

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0  2017-12-30
1  2017-12-31
2  2018-01-01
dtype: datetime64[ns]

>>> dates.dt.is_year_start
0   False
1   False
2    True
dtype: bool

>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')

>>> idx.is_year_start
array([False, False, True])
```
**pandas.Series.dt.is_year_end**

Series.dt.is_year_end

Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex  The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start  Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]

>>> dates.dt.is_year_end
0   False
1   True
2   False
dtype: bool
```

```python
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(["2017-12-30", "2017-12-31", "2018-01-01"],
dtype='datetime64[ns]', freq='D')
```

```python
>>> idx.is_year_end
array([False, True, False])
```

**pandas.Series.dt.is_leap_year**

Series.dt.is_leap_year

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

Returns

Series or ndarray  Booleans indicating if dates belong to a leap year.
Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
             dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False])
```

```python
>>> dates_series = pd.Series(idx)
>>> dates_series
0 2012-12-31
1 2013-12-31
2 2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0   True
1  False
2  False
dtype: bool
```

**pandas.Series.dt.daysinmonth**

Series.dt.daysinmonth

The number of days in the month.

**pandas.Series.dt.days_in_month**

Series.dt.days_in_month

The number of days in the month.

**pandas.Series.dt.tz**

Series.dt.tz

Return the timezone.

**Returns**

```
datetime.tzinfo, pytz.tzinfo.BaseTZInfo, dateutil.tz.tz.tzfile, or None  Returns None when the array is tz-naive.
```
pandas.Series.dt.freq

Series.dt.freq

Datetime methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.dt.to_period(*args, **kwargs)</td>
<td>Cast to PeriodArray/Index at a particular frequency.</td>
</tr>
<tr>
<td>Series.dt.to_pydatetime()</td>
<td>Return the data as an array of datetime.datetime objects.</td>
</tr>
<tr>
<td>Series.dt.tz_localize(*args, **kwargs)</td>
<td>Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.</td>
</tr>
<tr>
<td>Series.dt.tz_convert(*args, **kwargs)</td>
<td>Convert tz-aware Datetime Array/Index from one time zone to another.</td>
</tr>
<tr>
<td>Series.dt.normalize(*args, **kwargs)</td>
<td>Convert times to midnight.</td>
</tr>
<tr>
<td>Series.dt.strftime(*args, **kwargs)</td>
<td>Convert to Index using specified date_format.</td>
</tr>
<tr>
<td>Series.dt.round(*args, **kwargs)</td>
<td>Perform round operation on the data to the specified freq.</td>
</tr>
<tr>
<td>Series.dt.floor(*args, **kwargs)</td>
<td>Perform ceil operation on the data to the specified freq.</td>
</tr>
<tr>
<td>Series.dt.month_name(*args, **kwargs)</td>
<td>Return the month names of the DateTimeIndex with specified locale.</td>
</tr>
<tr>
<td>Series.dt.day_name(*args, **kwargs)</td>
<td>Return the day names of the DateTimeIndex with specified locale.</td>
</tr>
</tbody>
</table>

pandas.Series.dt.to_period

Series.dt.to_period(*args, **kwargs)

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

Parameters

freq [str or Offset, optional] One of pandas' offset strings or an Offset object. Will be inferred by default.

Returns

PeriodArray/Index

Raises

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

See also:

PeriodIndex Immutable ndarray holding ordinal values.
DatetimeIndex.to_pydatetime Return DatetimeIndex as object.
Examples

```python
>>> df = pd.DataFrame({"y": [1, 2, 3]},
                   index=pd.to_datetime(["2000-03-31 00:00:00",
                                         "2000-05-31 00:00:00",
                                         "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(["2000-03",
             "2000-05",
             "2000-08"],
            dtype='period[M]')
```

Infer the daily frequency

```python
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(["2017-01-01",
             "2017-01-02"],
            dtype='period[D]')
```

*pandas.Series.dt.to_pydatetime*

**Series.dt.to_pydatetime()**

Return the data as an array of `datetime.datetime` objects.

Timezone information is retained if present.

**Warning:** Python’s `datetime` uses microsecond resolution, which is lower than pandas (nanosecond). The values are truncated.

**Returns**

- `numpy.ndarray`: Object dtype array containing native Python datetime objects.

**See also:**

- `datetime.datetime`: Standard library value for a datetime.

**Examples**

```python
>>> s = pd.Series(pd.date_range('20180310', periods=2))
>>> s
0  2018-03-10
1  2018-03-11
dtype: datetime64[ns]
```

```python
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 11, 0, 0)], dtype=object)
```

*pandas’ nanosecond precision is truncated to microseconds.*

```python
>>> s = pd.Series(pd.date_range('20180310', periods=2, freq='ns'))
>>> s
(continues on next page)```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

0 2018-03-10 00:00:00.000000000
1 2018-03-10 00:00:00.000000001
dtype: datetime64[ns]

```python
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
      datetime.datetime(2018, 3, 10, 0, 0)], dtype=object)
```

**pandas.Series.dt.tz_localize**

`Series.dt.tz_localize(*args, **kwargs)`

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone.

This method can also be used to do the inverse – to create a time zone unaware object from an aware object. To that end, pass `tz=None`.

**Parameters**

- **tz** [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.
- **ambiguous** ['infer', ‘NaT’, bool array, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the `ambiguous` parameter dictates how ambiguous times should be handled.
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
- **nonexistent** ['shift_forward', 'shift_backward', ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return NaT where there are nonexistent times
  - timedelta objects will shift nonexistent times by the timedelta
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

- **Same type as self** Array/Index converted to the specified time zone.

**Raises**

- **TypeError** If the Datetime Array/Index is tz-aware and tz is not None.
See also:

**DatetimeIndex.tz_convert** Convert tz-aware DatetimeIndex from one time zone to another.

**Examples**

```python
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00', '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')

Localize DatetimeIndex in US/Eastern time zone:

```python
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00', '2018-03-02 09:00:00-05:00', '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

With the tz=None, we can remove the time zone information while keeping the local time (not converted to UTC):

```python
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00', '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```python
>>> s = pd.to_datetime(pd.Series(['2018-10-28 01:30:00',
                                 ...,
                                 '2018-10-28 02:00:00',
                                 '2018-10-28 02:30:00',
                                 '2018-10-28 02:00:00',
                                 '2018-10-28 02:30:00',
                                 '2018-10-28 03:00:00',
                                 '2018-10-28 03:30:00']))
```

```python
>>> s.dt.tz_localize('CET', ambiguous='infer')
0   2018-10-28 01:30:00+02:00
1   2018-10-28 02:00:00+02:00
2   2018-10-28 02:30:00+02:00
3   2018-10-28 02:00:00+01:00
4   2018-10-28 02:30:00+01:00
5   2018-10-28 03:00:00+01:00
6   2018-10-28 03:30:00+01:00
dtype: datetime64[ns, CET]
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarry to the ambiguous parameter to set the DST explicitly

```python
>>> s = pd.to_datetime(pd.Series(['2018-10-28 01:20:00',
                                 ...,
                                 '2018-10-28 02:36:00',
                                 (continues on next page)
```
If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or \texttt{\textquotesingle{}shift\_forward\textquotesingle{}} or \texttt{\textquotesingle{}shift\_backwards\textquotesingle{}}.

```python
>>> s = pd.to_datetime(pd.Series(['2015-03-29 02:30:00', '2015-03-29 03:30:00']))
```

```python
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
```

```python
0 2015-03-29 03:00:00+02:00
1 2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]
```

```python
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
```

```python
0 2015-03-29 01:59:59.999999999+01:00
1 2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]
```

```python
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
```

```python
0 2015-03-29 03:30:00+02:00
1 2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]
```

### Series.dt.tz_convert

**Series.dt.tz_convert(**\texttt{\textquotesingle{}args, **kwargs\textquotesingle{}}\)**

Convert tz-aware Datetime Array/Index from one time zone to another.

**Parameters**

- \texttt{tz} [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A \texttt{tz} of None will convert to UTC and remove the timezone information.

**Returns**

- Array or Index

**Raises**

- \texttt{TypeError} If Datetime Array/Index is tz-naive.

**See also:**

- \texttt{DatetimeIndex.tz} A timezone that has a variable offset from UTC.
- \texttt{DatetimeIndex.tz_localize} Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.
Examples

With the `tz` parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.date_range(start='2014-08-01 09:00',
                        freq='H', periods=3, tz='Europe/Berlin')

>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')

>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.date_range(start='2014-08-01 09:00', freq='H',
                      periods=3, tz='Europe/Berlin')

>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')

>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
              dtype='datetime64[ns]', freq='H')
```

`pandas.Series.dt.normalize`

Series.dt.normalize(*args, **kwargs)

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the
time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the .dt accessor, and directly on Datetime Ar-
ray/Index.

Returns

`DatetimeArray, DatetimeIndex or Series` The same type as the original data. Series will
have the same name and index. DatetimeIndex will have the same name.

See also:

`floor` Floor the datetimes to the specified freq.
**ceil** Ceil the datetimes to the specified freq.
**round** Round the datetimes to the specified freq.

### Examples

```python
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
                      ...       periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
                '2014-08-01 11:00:00+05:30',
                '2014-08-01 12:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
                '2014-08-01 00:00:00+05:30',
                '2014-08-01 00:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

**pandas.Series.dt.strftime**

Series.dt.strftime(*args, **kwargs)*

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

**Parameters**

- **date_format** [str] Date format string (e.g. “%Y-%m-%d”).

**Returns**

- **ndarray[object]** NumPy ndarray of formatted strings.

**See also:**

- **to_datetime** Convert the given argument to datetime.
- **DatetimeIndex.normalize** Return DatetimeIndex with times to midnight.
- **DatetimeIndex.round** Round the DatetimeIndex to the specified freq.
- **DatetimeIndex.floor** Floor the DatetimeIndex to the specified freq.

### Examples

```python
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
                      ...       periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```
pandas.Series.dt.round

Series.dt.round(*args, **kwargs)

Perform round operation on the data to the specified freq.

Parameters

- **freq** [str or Offset] The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.

- **ambiguous** ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

- **nonexistent** ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return NaT where there are nonexistent times
  - timedelta objects will shift nonexistent times by the timedelta
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns

- DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

- ValueError if the freq cannot be converted.

Notes

If the timestamps have a timezone, rounding will take place relative to the local (“wall”) time and re-localized to the same timezone. When rounding near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.
Examples

DatetimeIndex

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00', '2018-01-01 12:01:00'],
        dtype='datetime64[ns]', freq='T')
```

```python
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00', '2018-01-01 12:00:00'],
        dtype='datetime64[ns]', freq=None)
```

```python
Series

>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 03:30:00'], tz="Europe/Amsterdam")
```

```python
>>> rng_tz.floor("2H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
        dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

```python
>>> rng_tz.floor("2H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
        dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

**pandas.Series.dt.floor**

Series.dt.floor(*args, **kwargs)

Perform floor operation on the data to the specified freq.

**Parameters**

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.

ambiguous ['infer', bool-ndarray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:

- ‘infer’ will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- ‘NaT’ will return NaT where there are ambiguous times
- ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
nonexistent ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
- ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
- ‘NaT’ will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the freq cannot be converted.

Notes

If the timestamps have a timezone, flooring will take place relative to the local ("wall") time and re-localized to the same timezone. When flooring near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

Examples

DatetimeIndex

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
```
```
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
                '2018-01-01 12:01:00'], dtype='datetime64[ns]', freq='T')
```
```
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
                '2018-01-01 12:00:00'], dtype='datetime64[ns]', freq=None)
```

Series

```python
>>> pd.Series(rng).dt.floor("H")
```
```
0  2018-01-01 11:00:00
1  2018-01-01 12:00:00
2  2018-01-01 12:00:00
```
```
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> rng_tz = pd.DatetimeIndex(["2021-10-31 03:30:00"], tz="Europe/Amsterdam")
```
pandas.Series.dt.ceil

Series.dt.ceil(*args, **kwargs)
Perform ceil operation on the data to the specified freq.

Parameters
freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.
ambiguous ['infer', bool-narray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:
• ‘infer’ will attempt to infer fall dst-transition hours based on order
• bool-narray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
• ‘NaT’ will return NaT where there are ambiguous times
• ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
nonexistent ['shift_forward', 'shift_backward', ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
• ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
• ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
• ‘NaT’ will return NaT where there are nonexistent times
• timedelta objects will shift nonexistent times by the timedelta
• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns
DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises
ValueError if the freq cannot be converted.
Notes

If the timestamps have a timezone, ceiling will take place relative to the local ("wall") time and re-localized to the same timezone. When ceiling near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

Examples

DatetimeIndex

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
```
```python
>>> rng
```
```python
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00', '2018-01-01 12:01:00'], dtype='datetime64[ns]', freq='T')
```
```python
>>> rng.ceil('H')
```
```python
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00', '2018-01-01 13:00:00'], dtype='datetime64[ns]', freq=None)
```

Series

```python
>>> pd.Series(rng).dt.ceil("H")
```
```python
0 2018-01-01 12:00:00
1 2018-01-01 12:00:00
2 2018-01-01 13:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 01:30:00'], tz="Europe/Amsterdam")
```
```python
>>> rng_tz.ceil("H", ambiguous=False)
```
```python
DatetimeIndex(['2021-10-31 02:00:00+01:00'], dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```
```python
>>> rng_tz.ceil("H", ambiguous=True)
```
```python
DatetimeIndex(['2021-10-31 02:00:00+02:00'], dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

pandas.Series.dt.month_name

```python
Series.dt.month_name(*args, **kwargs)
```

Return the month names of the DateTimeIndex with specified locale.

Parameters

locale [str, optional] Locale determining the language in which to return the month name. Default is English locale.

Returns

Index Index of month names.
Examples

```python
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'],
               dtype='datetime64[ns]', freq='M')
```

```python
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

```
pandas.Series.dt.day_name
Series.dt.day_name(*args, **kwargs)
```

Return the day names of the DateTimeIndex with specified locale.

Parameters

- `locale` [str, optional] Locale determining the language in which to return the day name. Default is English locale.

Returns

- `Index` Index of day names.

Examples

```python
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03'],
               dtype='datetime64[ns]', freq='D')
```

```python
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

Period properties

- `Series.dt.qyear`
- `Series.dt.start_time`
- `Series.dt.end_time`
**pandas.Series.dt.qyear**

Series.dt.qyear

**pandas.Series.dt.start_time**

Series.dt.start_time

**pandas.Series.dt.end_time**

Series.dt.end_time

**Timedelta properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.dt.days</td>
<td>Number of days for each element.</td>
</tr>
<tr>
<td>Series.dt.seconds</td>
<td>Number of seconds (&gt;= 0 and less than 1 day) for each element.</td>
</tr>
<tr>
<td>Series.dt.microseconds</td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td>Series.dt.nanoseconds</td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</td>
</tr>
<tr>
<td>Series.dt.components</td>
<td>Return a Dataframe of the components of the Timedeltas.</td>
</tr>
</tbody>
</table>

**pandas.Series.dt.days**

Series.dt.days

  Number of days for each element.

**pandas.Series.dt.seconds**

Series.dt.seconds

  Number of seconds (>= 0 and less than 1 day) for each element.

**pandas.Series.dt.microseconds**

Series.dt.microseconds

  Number of microseconds (>= 0 and less than 1 second) for each element.
**pandas.Series.dt.nanoseconds**

Series.dt.nanoseconds
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

**pandas.Series.dt.components**

Series.dt.components
Return a Dataframe of the components of the Timedeltas.

Returns
DataFrame

**Examples**

```python
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='s'))
    s
0 0 days 00:00:00
1 0 days 00:00:01
2 0 days 00:00:02
3 0 days 00:00:03
4 0 days 00:00:04
dtype: timedelta64[ns]

>>> s.dt.components
    days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0      0       0        0       0             0            0            0
1      0       0        0       1             0            0            0
2      0       0        0       2             0            0            0
3      0       0        0       3             0            0            0
4      0       0        0       4             0            0            0
```

**Timedelta methods**

- `Series.dt.to_pytimedelta()` Return an array of native `datetime.timedelta` objects.
- `Series.dt.total_seconds(*args, **kwargs)` Return total duration of each element expressed in seconds.

**pandas.Series.dt.to_pytimedelta**

Series.dt.to_pytimedelta()
Return an array of native `datetime.timedelta` objects.

Python’s standard `datetime` library uses a different representation timedelta’s. This method converts a Series of pandas Timedeltas to `datetime.timedelta` format with the same length as the original Series.

Returns
`numpy.ndarray` Array of 1D containing data with `datetime.timedelta` type.

See also:
datetime.timedelta  A duration expressing the difference between two date, time, or datetime.

Examples

```python
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit="d"))
>>> s
dtype: timedelta64[ns]
```

```python
>>> s.dt.to_pytimedelta()
array([datetime.timedelta(0), datetime.timedelta(days=1),
      datetime.timedelta(days=2), datetime.timedelta(days=3),
      datetime.timedelta(days=4)], dtype=object)
```

pandas.Series.dt.total_seconds

Series.dt.total_seconds(*args, **kwargs)

Return total duration of each element expressed in seconds.

This method is available directly on TimedeltaArray, TimedeltaIndex and on Series containing timedelta values under the .dt namespace.

Returns

seconds  [[ndarray, Float64Index, Series]] When the calling object is a TimedeltaArray, the return type is ndarray. When the calling object is a TimedeltaIndex, the return type is a Float64Index. When the calling object is a Series, the return type is Series of type float64 whose index is the same as the original.

See also:

datetime.timedelta.total_seconds  Standard library version of this method.
TimedeltaIndex.components  Return a DataFrame with components of each Timedelta.

Examples

Series

```python
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
dtype: timedelta64[ns]
```
>>> s.dt.total_seconds()
    0   0.0
    1  86400.0
    2 172800.0
    3 259200.0
    4 345600.0
    dtype: float64

TimedeltaIndex

>>> idx = pd.to_timedelta(np.arange(5), unit='d')
>>> idx
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)

>>> idx.total_seconds()
Float64Index([0.0, 86400.0, 172800.0, 259200.0, 345600.0],
              dtype='float64')

String handling

Series.str can be used to access the values of the series as strings and apply several methods to it. These can be accessed like Series.str.<function/property>.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.str.capitalize()</td>
<td>Convert strings in the Series/Index to be capitalized.</td>
</tr>
<tr>
<td>Series.str.casefold()</td>
<td>Convert strings in the Series/Index to be casefolded.</td>
</tr>
<tr>
<td>Series.str.cat([others, sep, na_rep, join])</td>
<td>Concatenate strings in the Series/Index with given separator.</td>
</tr>
<tr>
<td>Series.str.center(width[, fillchar])</td>
<td>Pad left and right side of strings in the Series/Index.</td>
</tr>
<tr>
<td>Series.str.contains(pat[, case, flags, na, ...])</td>
<td>Test if pattern or regex is contained within a string of a Series or Index.</td>
</tr>
<tr>
<td>Series.str.count(pat[, flags])</td>
<td>Count occurrences of pattern in each string of the Series/Index.</td>
</tr>
<tr>
<td>Series.str.decode(encode, errors)</td>
<td>Decode character string in the Series/Index using indicated encoding.</td>
</tr>
<tr>
<td>Series.str.encode(encode, errors)</td>
<td>Encode character string in the Series/Index using indicated encoding.</td>
</tr>
<tr>
<td>Series.str.endswith(pat[, na])</td>
<td>Test if the end of each string element matches a pattern.</td>
</tr>
<tr>
<td>Series.str.extract(pat[, flags, expand])</td>
<td>Extract capture groups in the regex pat as columns in a DataFrame.</td>
</tr>
<tr>
<td>Series.str.extractall(pat[, flags])</td>
<td>Extract capture groups in the regex pat as columns in DataFrame.</td>
</tr>
<tr>
<td>Series.str.find(sub[, start, end])</td>
<td>Return lowest indexes in each strings in the Series/Index.</td>
</tr>
<tr>
<td>Series.str.findall(pat[, flags])</td>
<td>Find all occurrences of pattern or regular expression in the Series/Index.</td>
</tr>
<tr>
<td>Series.str.fullmatch(pat[, case, flags, na])</td>
<td>Determine if each string entirely matches a regular expression.</td>
</tr>
<tr>
<td>Series.str.get(i)</td>
<td>Extract element from each component at specified position.</td>
</tr>
<tr>
<td>Series.str.index(sub[, start, end])</td>
<td>Return lowest indexes in each string in Series/Index.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.str.join(sep)</code></td>
<td>Join lists contained as elements in the Series/Index with passed delimiter.</td>
</tr>
<tr>
<td><code>Series.str.len()</code></td>
<td>Compute the length of each element in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.ljust(width[, fillchar])</code></td>
<td>Pad left side of strings in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.lower()</code></td>
<td>Convert strings in the Series/Index to lowercase.</td>
</tr>
<tr>
<td><code>Series.str.lstrip([to_strip])</code></td>
<td>Remove leading characters.</td>
</tr>
<tr>
<td><code>Series.str.match(pat[, case, flags, na])</code></td>
<td>Determine if each string starts with a match of a regular expression.</td>
</tr>
<tr>
<td><code>Series.str.normalize(form)</code></td>
<td>Return the Unicode normal form for the strings in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.pad(width[, side, fillchar])</code></td>
<td>Pad strings in the Series/Index up to width.</td>
</tr>
<tr>
<td><code>Series.str.partition([sep, expand])</code></td>
<td>Split the string at the first occurrence of <code>sep</code>.</td>
</tr>
<tr>
<td><code>Series.str.removeprefix(prefix)</code></td>
<td>Remove a prefix from an object series.</td>
</tr>
<tr>
<td><code>Series.str.removesuffix(suffix)</code></td>
<td>Remove a suffix from an object series.</td>
</tr>
<tr>
<td><code>Series.str.replace(pat, repl[, n, case, ...])</code></td>
<td>Replace each occurrence of pattern/regex in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.rfind(sub[, start, end])</code></td>
<td>Return highest indexes in each strings in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.rindex(sub[, start, end])</code></td>
<td>Return highest indexes in each string in Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.rjust(width[, fillchar])</code></td>
<td>Pad left side of strings in the Series/Index.</td>
</tr>
<tr>
<td><code>Series.str.rpartition([sep, expand])</code></td>
<td>Split the string at the last occurrence of <code>sep</code>.</td>
</tr>
<tr>
<td><code>Series.str.rstrip([to_strip])</code></td>
<td>Remove trailing characters.</td>
</tr>
<tr>
<td><code>Series.str.slice([start, stop, step])</code></td>
<td>Slice substrings from each element in the Series or Index.</td>
</tr>
<tr>
<td><code>Series.str.slice_replace([start, stop, repl])</code></td>
<td>Replace a positional slice of a string with another value.</td>
</tr>
<tr>
<td><code>Series.str.split([pat, n, expand, regex])</code></td>
<td>Split strings around given separator/delimiter.</td>
</tr>
<tr>
<td><code>Series.str.rsplit([pat, n, expand])</code></td>
<td>Split strings around given separator/delimiter.</td>
</tr>
<tr>
<td><code>Series.str.startswith(pat[, na])</code></td>
<td>Test if the start of each string element matches a pattern.</td>
</tr>
<tr>
<td><code>Series.str.strip([to_strip])</code></td>
<td>Remove leading and trailing characters.</td>
</tr>
<tr>
<td><code>Series.str.swapcase()</code></td>
<td>Convert strings in the Series/Index to be swapcased.</td>
</tr>
<tr>
<td><code>Series.str.title()</code></td>
<td>Convert strings in the Series/Index to titlecase.</td>
</tr>
<tr>
<td><code>Series.str.translate(table)</code></td>
<td>Map all characters in the string through the given mapping table.</td>
</tr>
<tr>
<td><code>Series.str.upper()</code></td>
<td>Convert strings in the Series/Index to uppercase.</td>
</tr>
<tr>
<td><code>Series.str.wrap(width, **kwargs)</code></td>
<td>Wrap strings in Series/Index at specified line width.</td>
</tr>
<tr>
<td><code>Series.str.zfill(width)</code></td>
<td>Pad strings in the Series/Index by prepending '0' characters.</td>
</tr>
<tr>
<td><code>Series.str.isalnum()</code></td>
<td>Check whether all characters in each string are alphanumeric.</td>
</tr>
<tr>
<td><code>Series.str.isalpha()</code></td>
<td>Check whether all characters in each string are alphabetic.</td>
</tr>
<tr>
<td><code>Series.str.isdigit()</code></td>
<td>Check whether all characters in each string are digits.</td>
</tr>
<tr>
<td><code>Series.str.isspace()</code></td>
<td>Check whether all characters in each string are whitespace.</td>
</tr>
<tr>
<td><code>Series.str.islower()</code></td>
<td>Check whether all characters in each string are lowercase.</td>
</tr>
<tr>
<td><code>Series.str.isupper()</code></td>
<td>Check whether all characters in each string are uppercase.</td>
</tr>
<tr>
<td><code>Series.str.istitle()</code></td>
<td>Check whether all characters in each string are titlecase.</td>
</tr>
<tr>
<td><code>Series.str.isnumeric()</code></td>
<td>Check whether all characters in each string are numeric.</td>
</tr>
</tbody>
</table>

continues on next page
Scan and PDF conversion result:

Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.str.isdecimal()</td>
<td>Check whether all characters in each string are decimal.</td>
</tr>
<tr>
<td>Series.str.get_dummies(sep)</td>
<td>Return DataFrame of dummy/indicator variables for Series.</td>
</tr>
</tbody>
</table>

pandas.Series.str.capitalize

Series.str.capitalize()

Convert strings in the Series/Index to be capitalized.

Equivalent to str.capitalize().

Returns

Series or Index of object

See also:

- Series.str.lower  Converts all characters to lowercase.
- Series.str.upper  Converts all characters to uppercase.
- Series.str.title  Converts first character of each word to uppercase and remaining to lowercase.
- Series.str.capitalize Converts first character to uppercase and remaining to lowercase.
- Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.
- Series.str.casefold Removes all case distinctions in the string.

Examples

```python
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0    lower
1   CAPITALS
2  this is a sentence
3     SwApCaSe
dtype: object

>>> s.str.lower()
0    lower
1   capitals
2  this is a sentence
3    swapcase
dtype: object

>>> s.str.upper()
0    LOWER
1   CAPITALS
2  THIS IS A SENTENCE
3    SWAPCASE
dtype: object

>>> s.str.title()
0    Lower
1   Capitals
2  This Is A Sentence
```
pandas.Series.str.casefold

Series.str.casefold()

Convert strings in the Series/Index to be casefolded.

New in version 0.25.0.

Equivalent to str.casefold().

Returns

Series or Index of object

See also:

Series.str.lower Converts all characters to lowercase.
Series.str.upper Converts all characters to uppercase.
Series.str.title Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.
Series.str.casefold Removes all case distinctions in the string.

Examples

>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])

>>> s

0 lower
1 CAPITALS
2 this is a sentence
3 SwApCaSe
dtype: object

>>> s.str.lower()

0 lower
1 capitals

(continues on next page)
pandas.Series.str.cat

Series.str.cat(\texttt{\textit{others=\textit{None}, \textit{sep=\textit{None}, \textit{na_rep=\textit{None}, \textit{join=\textit{left}}})\}}

Concatenate strings in the Series/Index with given separator.

If \texttt{\textit{others}} is specified, this function concatenates the Series/Index and elements of \texttt{\textit{others}} element-wise. If \texttt{\textit{others}} is not passed, then all values in the Series/Index are concatenated into a single string with a given \texttt{\textit{sep}}.

**Parameters**

- **\texttt{\textit{others}}** [Series, Index, DataFrame, np.ndarray or list-like] Series, Index, DataFrame, np.ndarray (one- or two-dimensional) and other list-likes of strings must have the same length as the calling Series/Index, with the exception of indexed objects (i.e. Series/Index/DataFrame) if \texttt{\textit{join}} is not None.

If \texttt{\textit{others}} is a list-like that contains a combination of Series, Index or np.ndarray (1-dim), then all elements will be unpacked and must satisfy the above criteria individually.

If \texttt{\textit{others}} is None, the method returns the concatenation of all strings in the calling Series/Index.
sep [str, default ‘’] The separator between the different elements/columns. By default the empty string ‘’ is used.

na_rep [str or None, default None] Representation that is inserted for all missing values:

- If na_rep is None, and others is None, missing values in the Series/Index are omitted from the result.
- If na_rep is None, and others is not None, a row containing a missing value in any of the columns (before concatenation) will have a missing value in the result.

join [[‘left’, ‘right’, ‘outer’, ‘inner’], default ‘left’] Determines the join-style between the calling Series/Index and any Series/Index/DataFrame in others (objects without an index need to match the length of the calling Series/Index). To disable alignment, use .values on any Series/Index/DataFrame in others.

New in version 0.23.0.

Changed in version 1.0.0: Changed default of join from None to ‘left’.

Returns

str, Series or Index If others is None, str is returned, otherwise a Series/Index (same type as caller) of objects is returned.

See also:

split Split each string in the Series/Index.
join Join lists contained as elements in the Series/Index.

Examples

When not passing others, all values are concatenated into a single string:

```python
>>> s = pd.Series(['a', 'b', np.nan, 'd'])
>>> s.str.cat(sep=' ')
'a b d'
```

By default, NA values in the Series are ignored. Using na_rep, they can be given a representation:

```python
>>> s.str.cat(sep=' ', na_rep='?')
'a b ? d'
```

If others is specified, corresponding values are concatenated with the separator. Result will be a Series of strings.

```python
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',')
0   a,A
1   b,B
2   NaN
3   d,D
dtype: object
```

Missing values will remain missing in the result, but can again be represented using na_rep.

```python
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',' , na_rep='-' )
0   a,A
1   b,B
2   -,C
```

(continues on next page)
If `sep` is not specified, the values are concatenated without separation.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], na_rep='--')
0  aA
1  bB
2  -C
3  dD
dtype: object
```

Series with different indexes can be aligned before concatenation. The `join`-keyword works as in other methods.

```
>>> t = pd.Series(['d', 'a', 'e', 'c'], index=[3, 0, 4, 2])
>>> s.str.cat(t, join='left', na_rep='--')
0  aa
1  b-
2  -c
3  dd
dtype: object
>>> s.str.cat(t, join='outer', na_rep='--')
0  aa
1  b-
2  -c
3  dd
4  -e
dtype: object
>>> s.str.cat(t, join='inner', na_rep='--')
0  aa
2  -c
3  dd
dtype: object
>>> s.str.cat(t, join='right', na_rep='--')
3  dd
0  aa
4  -e
2  -c
dtype: object
```

For more examples, see [here](#).
pandas.Series.str.center

Series.str.center(width, fillchar='')

Pad left and right side of strings in the Series/Index.
Equivalent to str.center().

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with fillchar.
fillchar [str] Additional character for filling, default is whitespace.

Returns

filled [Series/Index of objects.]

pandas.Series.str.contains

Series.str.contains(pat, case=True, flags=0, na=None, regex=True)

Test if pattern or regex is contained within a string of a Series or Index.
Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

Parameters

pat [str] Character sequence or regular expression.
case [bool, default True] If True, case sensitive.
flags [int, default 0 (no flags)] Flags to pass through to the re module, e.g. re.IGNORECASE.
na [scalar, optional] Fill value for missing values. The default depends on dtype of the array.
For object-dtype, numpy.nan is used. For StringDtype, pandas.NA is used.
regex [bool, default True] If True, assumes the pat is a regular expression.
If False, treats the pat as a literal string.

Returns

Series or Index of boolean values A Series or Index of boolean values indicating whether the given pattern is contained within the string of each element of the Series or Index.

See also:
match Analogous, but stricter, relying on re.match instead of re.search.
Series.str.startswith Test if the start of each string element matches a pattern.
Series.str.endswith Same as startswith, but tests the end of string.

Examples

Returning a Series of booleans using only a literal pattern.

```python
>>> s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s1.str.contains('og', regex=False)
0    False
1    True
2    False
Name: 0, dtype: bool
```
Returning an Index of booleans using only a literal pattern.

```python
ind = pd.Index(["Mouse", 'dog', 'house and parrot', '23.0', np.Nan])
ind.str.contains('23', regex=False)
Index([False, False, False, True, nan], dtype='object')
```

Specifying case sensitivity using `case`.

```python
s1.str.contains('oG', case=True, regex=True)
0   False
1   False
2   False
3   False
4    NaN
dtype: object
```

Specifying `na` to be `False` instead of `NaN` replaces `NaN` values with `False`. If Series or Index does not contain `NaN` values the resultant dtype will be `bool`, otherwise, an `object` dtype.

```python
s1.str.contains('og', na=False, regex=True)
0   False
1    True
2   False
3   False
4   False
dtype: bool
```

Returning ‘house’ or ‘dog’ when either expression occurs in a string.

```python
s1.str.contains('house|dog', regex=True)
0   False
1    True
2    True
3   False
4    NaN
dtype: object
```

Ignoring case sensitivity using `flags` with regex.

```python
import re
s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0   False
1   False
2    True
3   False
4    NaN
dtype: object
```

Returning any digit using regular expression.
Ensure \textit{pat} is a not a literal pattern when \texttt{regex} is set to True. Note in the following example one might expect only \texttt{s2[1]} and \texttt{s2[3]} to return \texttt{True}. However, \texttt{'.0'} as a regex matches any character followed by a \texttt{0}.

\begin{verbatim}
>>> s2 = pd.Series(['40', '40.0', '41', '41.0', '35'])
>>> s2.str.contains('.0', regex=True)
0   True
1   True
2  False
3   True
4  False
dtype: bool
\end{verbatim}

\texttt{pandas.Series.str.count}

\texttt{Series.str.count(pat, flags=0)}

Count occurrences of pattern in each string of the Series/Index.

This function is used to count the number of times a particular regex pattern is repeated in each of the string elements of the \texttt{Series}.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{pat} [str] Valid regular expression.
  \item \texttt{flags} [int, default 0, meaning no flags] Flags for the \texttt{re} module. For a complete list, see here.
  \item **\texttt{kwargs}** For compatibility with other string methods. Not used.
\end{itemize}

\textbf{Returns}

\texttt{Series or Index} Same type as the calling object containing the integer counts.

\textbf{See also:}

\begin{itemize}
  \item \texttt{re} Standard library module for regular expressions.
  \item \texttt{str.count} Standard library version, without regular expression support.
\end{itemize}

\textbf{Notes}

Some characters need to be escaped when passing in \texttt{pat}. eg. \texttt{"\$"} has a special meaning in regex and must be escaped when finding this literal character.
Examples

```python
>>> s = pd.Series(['A', 'B', 'Aaba', 'Baca', np.nan, 'CABA', 'cat'])
>>> s.str.count('a')
0    0.0
1    0.0
2    2.0
3    2.0
4   NaN
5    0.0
6    1.0
dtype: float64
```

Escape '$' to find the literal dollar sign.

```python
>>> s = pd.Series(['$', 'B', 'Aab$', '$$ca', 'C$B$', 'cat'])
>>> s.str.count(r'\$')
0    1
1    0
2    1
3    2
4    2
5    0
dtype: int64
```

This is also available on Index

```python
>>> pd.Index(['A', 'A', 'Aaba', 'cat']).str.count('a')
Int64Index([0, 0, 2, 1], dtype='int64')
```

**pandas.Series.str.decode**

Series.str.decode(encoding, errors='strict')

Decode character string in the Series/Index using indicated encoding.

Equivalent to str.decode() in python2 and bytes.decode() in python3.

**Parameters**

- **encoding** [str]
- **errors** [str, optional]

**Returns**

Series or Index
pandas.Series.str.encode

Series.str.encode(encoding, errors='strict')

Encode character string in the Series/Index using indicated encoding.
Equivalent to str.encode().

Parameters
encoding [str]
errors [str, optional]

Returns
encoded [Series/Index of objects]

pandas.Series.str.endswith

Series.str.endswith(pat, na=None)

Test if the end of each string element matches a pattern.
Equivalent to str.endswith().

Parameters
pat [str] Character sequence. Regular expressions are not accepted.
na [object, default NaN] Object shown if element tested is not a string. The default depends on dtype of the array. For object-dtype, numpy.nan is used. For StringDtype, pandas.NA is used.

Returns
Series or Index of bool A Series of booleans indicating whether the given pattern matches the end of each string element.

See also:
str.endswith Python standard library string method.
Series.str.startswith Same as endswith, but tests the start of string.
Series.str.contains Tests if string element contains a pattern.

Examples

```python
>>> s = pd.Series(['bat', 'bear', 'caT', np.nan])
>>> s
0   bat
1   bear
2   caT
3    NaN
dtype: object

>>> s.str.endswith('t')
0   True
1   False
2   False
3    NaN
dtype: object
```
Specifying `na` to be `False` instead of `NaN`.

```python
>>> s.str.endswith('t', na=False)
0   True
1   False
2   False
3   False
dtype: bool
```

### pandas.Series.str.extract

`Series.str.extract(pat, flags=0, expand=True)`

Extract capture groups in the regex `pat` as columns in a DataFrame.

For each subject string in the Series, extract groups from the first match of regular expression `pat`.

**Parameters**

- `pat` [str] Regular expression pattern with capturing groups.
- `flags` [int, default 0 (no flags)] Flags from the `re` module, e.g. `re.IGNORECASE`, that modify regular expression matching for things like case, spaces, etc. For more details, see `re`.
- `expand` [bool, default True] If True, return DataFrame with one column per capture group. If False, return a Series/Index if there is one capture group or DataFrame if there are multiple capture groups.

**Returns**

- **DataFrame or Series or Index** A DataFrame with one row for each subject string, and one column for each group. Any capture group names in regular expression `pat` will be used for column names; otherwise capture group numbers will be used. The dtype of each result column is always object, even when no match is found. If `expand=False` and `pat` has only one capture group, then return a Series (if subject is a Series) or Index (if subject is an Index).

**See also:**

- `extractall` Returns all matches (not just the first match).

**Examples**

A pattern with two groups will return a DataFrame with two columns. Non-matches will be NaN.

```python
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.extract(r'([ab])(\d)')
   0  1
0  a  1
1  b  2
2  NaN NaN
```

A pattern may contain optional groups.

```python
>>> s.str.extract(r'([ab])?(\d)')
   0  1
0  a  1
```
Named groups will become column names in the result.

```python
>>> s.str.extract(r'(\P<letter>ab)(?P<digit>\d)\')
letter digit
0    a  1
1    b  2
2  NaN NaN
```

A pattern with one group will return a DataFrame with one column if `expand=True`.

```python
>>> s.str.extract(r'(ab)(\d)', expand=True)
 0
0   1
1   2
2  NaN
```

A pattern with one group will return a Series if `expand=False`.

```python
>>> s.str.extract(r'(ab)(\d)', expand=False)
 0 1
0 1
1 2
2 NaN
dtype: object
```

### pandas.Series.str.extractall

**Series.str.extractall** *(pat, flags=0)*

Extract capture groups in the regex `pat` as columns in DataFrame.

For each subject string in the Series, extract groups from all matches of regular expression `pat`. When each subject string in the Series has exactly one match, `extractall(pat).xs(0, level='match')` is the same as `extract(pat)`.  

**Parameters**

- `pat` [str] Regular expression pattern with capturing groups.
- `flags` [int, default 0 (no flags)] A `re` module flag, for example `re.IGNORECASE`. These allow to modify regular expression matching for things like case, spaces, etc. Multiple flags can be combined with the bitwise OR operator, for example `re.IGNORECASE | re.MULTILINE`.

**Returns**

- `DataFrame` A DataFrame with one row for each match, and one column for each group. Its rows have a `MultiIndex` with first levels that come from the subject Series. The last level is named ‘match’ and indexes the matches in each item of the Series. Any capture group names in regular expression `pat` will be used for column names; otherwise capture group numbers will be used.

**See also:**

- `extract` Returns first match only (not all matches).
Examples

A pattern with one group will return a DataFrame with one column. Indices with no matches will not appear in the result.

```python
>>> s = pd.Series(["a1", "b1", "c1"], index=["A", "B", "C"])
>>> s.str.extractall(r"[ab](\d)")
match
 0
   A  1
   1  2
   B  1
```

Capture group names are used for column names of the result.

```python
>>> s.str.extractall(r"[ab](?P<digit>\d)")
```

A pattern with two groups will return a DataFrame with two columns.

```python
>>> s.str.extractall(r"(?P<letter>[ab])(?P<digit>\d)")
```

Optional groups that do not match are NaN in the result.

```python
>>> s.str.extractall(r"(?P<letter>[ab])?(?P<digit>\d)")
```

pandas.Series.str.find

Series.str.find(sub, start=0, end=None)

Return lowest indexes in each strings in the Series/Index.

Each of returned indexes corresponds to the position where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard str.find().

Parameters

- sub [str] Substring being searched.
- start [int] Left edge index.
- end [int] Right edge index.
Returns

Series or Index of int.

See also:

rfind Return highest indexes in each strings.

**pandas.Series.str.findall**

Series.str.findall(*pat*, *flags=0*)

Find all occurrences of pattern or regular expression in the Series/Index.

Equivalent to applying re.findall() to all the elements in the Series/Index.

**Parameters**

- *pat* [str] Pattern or regular expression.
- *flags* [int, default 0] Flags from re module, e.g. re.IGNORECASE (default is 0, which means no flags).

**Returns**

Series/Index of lists of strings All non-overlapping matches of pattern or regular expression in each string of this Series/Index.

See also:

count Count occurrences of pattern or regular expression in each string of the Series/Index.
extractall For each string in the Series, extract groups from all matches of regular expression and return a DataFrame with one row for each match and one column for each group.
re.findall The equivalent re function to all non-overlapping matches of pattern or regular expression in string, as a list of strings.

**Examples**

```python
>>> s = pd.Series(['Lion', 'Monkey', 'Rabbit'])
```

The search for the pattern ‘Monkey’ returns one match:

```python
>>> s.str.findall('Monkey')
0    []
1  [Monkey]
2    []
dtype: object
```

On the other hand, the search for the pattern ‘MONKEY’ doesn’t return any match:

```python
>>> s.str.findall('MONKEY')
0    []
1    []
2    []
dtype: object
```

Flags can be added to the pattern or regular expression. For instance, to find the pattern ‘MONKEY’ ignoring the case:
```python
>>> import re
>>> s.str.findall('MONKEY', flags=re.IGNORECASE)
0  []
1  [Monkey]
2  []
dtype: object
```

When the pattern matches more than one string in the Series, all matches are returned:

```python
>>> s.str.findall('on')
0  [on]
1  [on]
2  []
dtype: object
```

Regular expressions are supported too. For instance, the search for all the strings ending with the word ‘on’ is shown next:

```python
>>> s.str.findall('on$')
0  [on]
1  []
2  []
dtype: object
```

If the pattern is found more than once in the same string, then a list of multiple strings is returned:

```python
>>> s.str.findall('b')
0  []
1  []
2  [b, b]
dtype: object
```

**pandas.Series.str.fullmatch**

Series.str.fullmatch(*pat, case=True, flags=0, na=None*)

Determine if each string entirely matches a regular expression.

New in version 1.1.0.

**Parameters**

- **pat** [str] Character sequence or regular expression.
- **case** [bool, default True] If True, case sensitive.
- **flags** [int, default 0 (no flags)] Regex module flags, e.g. re.IGNORECASE.
- **na** [scalar, optional] Fill value for missing values. The default depends on dtype of the array. For object-dtype, numpy.nan is used. For StringDtype, pandas.NA is used.

**Returns**

Series/Index/array of boolean values

See also:

- **match** Similar, but also returns True when only a prefix of the string matches the regular expression.
- **extract** Extract matched groups.
pandas.Series.str.get

Series.str.get(i)

Extract element from each component at specified position.

Extract element from lists, tuples, or strings in each element in the Series/Index.

Parameters

i [int] Position of element to extract.

Returns

Series or Index

Examples

```python
>>> s = pd.Series(["String",
... (1, 2, 3),
... ["a", "b", "c"],
... 123,
... -456,
... {1: "Hello", "2": "World"}])
>>> s
0    String
1      (1, 2, 3)
2      [a, b, c]
3       123
4       -456
5    {1: 'Hello', '2': 'World'}
dtype: object
```

```python
>>> s.str.get(1)
0    t
1    2
2    b
3   NaN
4   NaN
5    Hello
dtype: object
```

```python
>>> s.str.get(-1)
0    g
1    3
2    c
3   NaN
4   NaN
5   None
dtype: object
```
**pandas.Series.str.index**

Series.str.index(sub, start=0, end=None)

Return lowest indexes in each string in Series/Index.

Each of the returned indexes corresponds to the position where the substring is fully contained between [start:end]. This is the same as str.find except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard str.index.

**Parameters**
- sub [str] Substring being searched.
- start [int] Left edge index.
- end [int] Right edge index.

**Returns**
Series or Index of object

**See also:**
- **rindex** Return highest indexes in each strings.

**pandas.Series.str.join**

Series.str.join(sep)

Join lists contained as elements in the Series/Index with passed delimiter.

If the elements of a Series are lists themselves, join the content of these lists using the delimiter passed to the function. This function is an equivalent to str.join().

**Parameters**
- sep [str] Delimiter to use between list entries.

**Returns**
Series/Index: object The list entries concatenated by intervening occurrences of the delimiter.

**Raises**
- AttributeError If the supplied Series contains neither strings nor lists.

**See also:**
- str.join Standard library version of this method.
- Series.str.split Split strings around given separator/delimiter.

**Notes**

If any of the list items is not a string object, the result of the join will be NaN.
Examples

Example with a list that contains non-string elements.

```python
>>> s = pd.Series([['lion', 'elephant', 'zebra'],
                  [1.1, 2.2, 3.3],
                  ['cat', np.nan, 'dog'],
                  ['cow', 4.5, 'goat'],
                  ['duck', ['swan', 'fish', 'guppy']]])
```

Join all lists using a ‘-’. The lists containing object(s) of types other than str will produce a NaN.

```python
>>> s.str.join('-')
0    lion-elephant-zebra
1     NaN
2     NaN
3     NaN
4     NaN
```

### pandas.Series.str.len

Series.str.len()

Compute the length of each element in the Series/Index.

The element may be a sequence (such as a string, tuple or list) or a collection (such as a dictionary).

**Returns**

- **Series or Index of int**: A Series or Index of integer values indicating the length of each element in the Series or Index.

**See also:**

- `str.len`: Python built-in function returning the length of an object.
- `Series.size`: Returns the length of the Series.

**Examples**

Returns the length (number of characters) in a string. Returns the number of entries for dictionaries, lists or tuples.

```python
>>> s = pd.Series(['dog',
                  '',
                  5,
                  {'foo': 'bar'},
                  [2, 3, 5, 7],
                  ('one', 'two', 'three')])
```
pandas.Series.str.ljust

Series.str.ljust(width, fillchar=' ')  
Pad right side of strings in the Series/Index.  
Equivalent to str.ljust().  
Parameters

width [int] Minimum width of resulting string; additional characters will be filled with fillchar.  
fillchar [str] Additional character for filling, default is whitespace.  
Returns

filled [Series/Index of objects.]

pandas.Series.str.lower

Series.str.lower()  
Convert strings in the Series/Index to lowercase.  
Equivalent to str.lower().  
Returns

Series or Index of object

See also:

Series.str.lower  Converts all characters to lowercase.  
Series.str.upper  Converts all characters to uppercase.  
Series.str.title  Converts first character of each word to uppercase and remaining to lowercase.  
Series.str.capitalize  Converts first character to uppercase and remaining to lowercase.  
Series.str.swapcase  Converts uppercase to lowercase and lowercase to uppercase.  
Series.str.casefold  Removes all case distinctions in the string.
Examples

```python
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0    lower
1    CAPITALS
2    this is a sentence
3    SwApCaSe
dtype: object

>>> s.str.lower()
0    lower
1    capitals
2    this is a sentence
3    swapcase
dtype: object

>>> s.str.upper()
0    LOWER
1    CAPITALS
2    THIS IS A SENTENCE
3    SWAPCASE
dtype: object

>>> s.str.title()
0    Lower
1    Capitals
2    This Is A Sentence
3    Swapcase
dtype: object

>>> s.str.capitalize()
0    lower
1    Capitals
2    This is a sentence
3    Swapcase
dtype: object

>>> s.str.swapcase()
0    LOWER
1    capitals
2    THIS IS A SENTENCE
3    sWaPcAsE
dtype: object
```
pandas.Series.str.lstrip

Series.str.lstrip(to_strip=None)
Remove leading characters.
Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left side. Equivalent to \texttt{str.lstrip()}.

**Parameters**

to_strip [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

**Returns**

Series or Index of object

See also:

Series.str.strip Remove leading and trailing characters in Series/Index.
Series.str.lstrip Remove leading characters in Series/Index.
Series.str.rstrip Remove trailing characters in Series/Index.

**Examples**

```python
>>> s = pd.Series(['1. Ant.', '2. Bee!\n', '3. Cat?\t', np.nan])
>>> s
0     1. Ant.
1    2. Bee!
2   3. Cat?
3    NaN
dtype: object

>>> s.str.strip()
0     1. Ant.
1    2. Bee!
2   3. Cat?
3    NaN
dtype: object

>>> s.str.lstrip('123.')
0    Ant.
1   Bee!
2   Cat?
3    NaN
dtype: object

>>> s.str.rstrip('.!? \n\t')
0    1. Ant
1    2. Bee
2    3. Cat
3    NaN
dtype: object
```
```python
>>> s.str.strip('123.!? \n\t')
0    Ant
1    Bee
2    Cat
3  NaN
dtype: object
```

**pandas.Series.str.match**

```python
Series.str.match(pat, case=True, flags=0, na=None)
```

Determine if each string starts with a match of a regular expression.

**Parameters**

- **pat** [str] Character sequence or regular expression.
- **case** [bool, default True] If True, case sensitive.
- **flags** [int, default 0 (no flags)] Regex module flags, e.g. re.IGNORECASE.
- **na** [scalar, optional] Fill value for missing values. The default depends on dtype of the array. For object-dtype, numpy.nan is used. For StringDtype, pandas.NA is used.

**Returns**

Series/Index/array of boolean values

See also:

- `fullmatch` Stricter matching that requires the entire string to match.
- `contains` Analogous, but less strict, relying on re.search instead of re.match.
- `extract` Extract matched groups.

**pandas.Series.str.normalize**

```python
Series.str.normalize(form)
```

Return the Unicode normal form for the strings in the Series/Index.

For more information on the forms, see the `unicodedata.normalize()`.

**Parameters**

- **form** [{'NFC', 'NFKC', 'NFD', 'NFKD'}] Unicode form.

**Returns**

normalized [Series/Index of objects]

**pandas.Series.str.pad**

```python
Series.str.pad(width, side='left', fillchar='')
```

Pad strings in the Series/Index up to width.

**Parameters**

- **width** [int] Minimum width of resulting string; additional characters will be filled with character defined in fillchar.
- **side** [{'left', 'right', 'both'}, default 'left'] Side from which to fill resulting string.
fillchar  [str, default ‘ ‘] Additional character for filling, default is whitespace.

Returns

Series or Index of object  Returns Series or Index with minimum number of char in object.

See also:

Series.str.rjust  Fills the left side of strings with an arbitrary character. Equivalent to Series.str.pad(side='left')

Series.str.ljust  Fills the right side of strings with an arbitrary character. Equivalent to Series.str.pad(side='right')

Series.str.center  Fills both sides of strings with an arbitrary character. Equivalent to Series.str.pad(side='both')

Series.str.zfill  Pad strings in the Series/Index by prepending '0' character. Equivalent to Series.str.pad(side='left', fillchar='0')

Examples

```python
>>> s = pd.Series(['caribou', 'tiger'])
>>> s
0 caribou
1 tiger
dtype: object

>>> s.str.pad(width=10)
0 caribou
1 tiger
dtype: object

>>> s.str.pad(width=10, side='right', fillchar='-')
0 caribou---
1 tiger-----
dtype: object

>>> s.str.pad(width=10, side='both', fillchar='-')
0 --caribou--
1 --tiger---
dtype: object
```

pandas.Series.str.partition

Series.str.partition(sep='.', expand=True)

Split the string at the first occurrence of sep.

This method splits the string at the first occurrence of sep, and returns 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing the string itself, followed by two empty strings.

Parameters

sep  [str, default whitespace] String to split on.

expand  [bool, default True] If True, return DataFrame/MultiIndex expanding dimensionality. If False, return Series/Index.
Returns

DataFrame/MultiIndex or Series/Index of objects

See also:

rpartition Split the string at the last occurrence of sep.
Series.str.split Split strings around given separators.
str.partition Standard library version.

Examples

```python
>>> s = pd.Series(['Linda van der Berg', 'George Pitt-Rivers'])
>>> s
0   Linda van der Berg
1   George Pitt-Rivers
dtype: object

>>> s.str.partition()
     0   1   2
0  Linda  van  Berg
1   George  Pitt-Rivers

To partition by the last space instead of the first one:

```python
>>> s.str.rpartition()
     0   1   2
0  Linda  ,   Berg
1   George  Pitt-Rivers

To partition by something different than a space:

```python
>>> s.str.partition('-')
     0   1   2
0  Linda  van  Berg
1   George  Pitt  Rivers

To return a Series containing tuples instead of a DataFrame:

```python
>>> s.str.partition('-', expand=False)
0   (Linda van der Berg, , )
1   (George Pitt, -, Rivers)
dtype: object

Also available on indices:

```python
>>> idx = pd.Index(['X 123', 'Y 999'])
>>> idx
Index(['X 123', 'Y 999'], dtype='object')

Which will create a MultiIndex:

```python
>>> idx.str.partition()
MultiIndex([('X', , '123'),
	('Y', , '999')])
```

(continues on next page)
Or an index with tuples with `expand=False`:

```
>>> idx.str.partition(expand=False)
Index([('X', ' ', '123'), ('Y', ' ', '999')], dtype='object')
```

### pandas.Series.str.removeprefix

Series.str.removeprefix(prefix)
Remove a prefix from an object series. If the prefix is not present, the original string will be returned.

**Parameters**

prefix [str] Remove the prefix of the string.

**Returns**

Series/Index: object  The Series or Index with given prefix removed.

See also:

Series.str.removesuffix  Remove a suffix from an object series.

**Examples**

```
>>> s = pd.Series(['str_foo', 'str_bar', 'no_prefix'])

```

```
>>> s
0 str_foo
1 str_bar
2 no prefix

dtype: object
```

```
>>> s.str.removeprefix('str_')
0 foo
1 bar
2 no_prefix

dtype: object
```

```
>>> s = pd.Series(['foo_str', 'bar_str', 'no_suffix'])
```

```
>>> s
0 foo_str
1 bar_str
2 no_suffix

dtype: object
```

```
>>> s.str.removesuffix('_str')
0 foo
1 bar
2 no_suffix

dtype: object
```
pandas.Series.str.removesuffix

Series.str.removesuffix(suffix)
Remove a suffix from an object series. If the suffix is not present, the original string will be returned.

Parameters

suffix [str] Remove the suffix of the string.

Returns

Series/Index: object The Series or Index with given suffix removed.

See also:

Series.str.removeprefix Remove a prefix from an object series.

Examples

```python
>>> s = pd.Series(["str_foo", "str_bar", "no_prefix"])
>>> s
0  str_foo
1  str_bar
2  no_prefix
dtype: object

>>> s.str.removeprefix("str_")
0   foo
1   bar
2  no_prefix
dtype: object
```

```python
>>> s = pd.Series(["foo_str", "bar_str", "no_suffix"])
>>> s
0  foo_str
1  bar_str
2  no_suffix
dtype: object

>>> s.str.removesuffix("_str")
0   foo
1   bar
2  no_suffix
dtype: object
```

pandas.Series.str.repeat

Series.str.repeat(repeats)
Duplicate each string in the Series or Index.

Parameters

repeats [int or sequence of int] Same value for all (int) or different value per (sequence).

Returns

Series or Index of object Series or Index of repeated string objects specified by input parameter repeats.
Examples

```python
>>> s = pd.Series(['a', 'b', 'c'])
```
```
>>> s
0   a
1   b
2   c
dtype: object
```

Single int repeats string in Series

```python
>>> s.str.repeat(repeats=2)
```
```
0   aa
1   bb
2   cc
dtype: object
```

Sequence of int repeats corresponding string in Series

```python
>>> s.str.repeat(repeats=[1, 2, 3])
```
```
0   a
1   bb
2   ccc
dtype: object
```

**pandas.Series.str.replace**

Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=None)

Replace each occurrence of pattern/regex in the Series/Index.

Equivalent to str.replace() or re.sub(), depending on the regex value.

**Parameters**

- **pat**  [str or compiled regex] String can be a character sequence or regular expression.
- **repl**  [str or callable] Replacement string or a callable. The callable is passed the regex match object and must return a replacement string to be used. See re.sub().
- **n**  [int, default -1 (all)] Number of replacements to make from start.
- **case**  [bool, default None] Determines if replace is case sensitive:
  - If True, case sensitive (the default if pat is a string)
  - Set to False for case insensitive
  - Cannot be set if pat is a compiled regex.
- **flags**  [int, default 0 (no flags)] Regex module flags, e.g. re.IGNORECASE. Cannot be set if pat is a compiled regex.
- **regex**  [bool, default True] Determines if the passed-in pattern is a regular expression:
  - If True, assumes the passed-in pattern is a regular expression.
  - If False, treats the pattern as a literal string
  - Cannot be set to False if pat is a compiled regex or repl is a callable.
New in version 0.23.0.

Returns

**Series or Index of object** A copy of the object with all matching occurrences of *pat* replaced by *repl*.

Raises

**ValueError**

- if *regex* is False and *repl* is a callable or *pat* is a compiled regex
- if *pat* is a compiled regex and *case* or *flags* is set

Notes

When *pat* is a compiled regex, all flags should be included in the compiled regex. Use of *case*, *flags*, or *regex=False* with a compiled regex will raise an error.

Examples

When *pat* is a string and *regex* is True (the default), the given *pat* is compiled as a regex. When *repl* is a string, it replaces matching regex patterns as with *re.sub()*. NaN value(s) in the Series are left as is:

```python
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f.', 'ba', regex=True)
0   bao
1   baz
2  NaN
dtype: object
```

When *pat* is a string and *regex* is False, every *pat* is replaced with *repl* as with *str.replace()::

```python
>>> pd.Series(['f.o', 'fuz', np.nan]).str.replace('f.', 'ba', regex=False)
0   bao
1   fuz
2  NaN
dtype: object
```

When *repl* is a callable, it is called on every *pat* using *re.sub()* The callable should expect one positional argument (a regex object) and return a string.

To get the idea:

```python
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f', repr, regex=True)
0  <re.Match object; span=(0, 1), match='f'>oo
1  <re.Match object; span=(0, 1), match='f'>uz
2  NaN
dtype: object
```

Reverse every lowercase alphabetic word:

```python
>>> repl = lambda m: m.group(0)[::-1]
>>> ser = pd.Series(['foo 123', 'bar baz', np.nan])
>>> ser.str.replace(r'[a-z]+', repl, regex=True)
0   oof 123
```

(continues on next page)
Using regex groups (extract second group and swap case):

```python
>>> pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
>>> repl = lambda m: m.group('two').swapcase()
>>> ser = pd.Series(['One Two Three', 'Foo Bar Baz'])
>>> ser.str.replace(pat, repl, regex=True)
0 tWO
1 bAR
dtype: object
```

Using a compiled regex with flags

```python
>>> import re
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar', regex=True)
0 foo
1 bar
2 NaN
dtype: object
```

**pandas.Series.str.rfind**

Series.str.rfind(sub, start=0, end=None)

Return highest indexes in each strings in the Series/Index.

Each of returned indexes corresponds to the position where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard str.rfind().

Parameters

- **sub** [str] Substring being searched.
- **start** [int] Left edge index.
- **end** [int] Right edge index.

Returns

Series or Index of int.

See also:

- **find** Return lowest indexes in each strings.
**pandas.Series.str.rindex**

Series.str.rindex(sub, start=0, end=None)

Return highest indexes in each string in Series/Index.

Each of the returned indexes corresponds to the position where the substring is fully contained between [start:end]. This is the same as str.rfind except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard str.rindex.

**Parameters**

- sub [str] Substring being searched.
- start [int] Left edge index.
- end [int] Right edge index.

**Returns**

Series or Index of object

See also:

- index Return lowest indexes in each strings.

**pandas.Series.str.rjust**

Series.str.rjust(width, fillchar=' ')

Pad left side of strings in the Series/Index.

Equivalent to str.rjust().

**Parameters**

- width [int] Minimum width of resulting string; additional characters will be filled with fillchar.
- fillchar [str] Additional character for filling, default is whitespace.

**Returns**

filled [Series/Index of objects.]

**pandas.Series.str.rpartition**

Series.str.rpartition(sep=' ', expand=True)

Split the string at the last occurrence of sep.

This method splits the string at the last occurrence of sep, and returns 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing two empty strings, followed by the string itself.

**Parameters**

- sep [str, default whitespace] String to split on.
- expand [bool, default True] If True, return DataFrame/MultiIndex expanding dimensionality.
  
  If False, return Series/Index.

**Returns**

DataFrame/MultiIndex or Series/Index of objects

See also:
**partition**  Split the string at the first occurrence of *sep*.

**Series.str.split**  Split strings around given separators.

**str.partition**  Standard library version.

### Examples

```python
>>> s = pd.Series(['Linda van der Berg', 'George Pitt-Rivers'])
>>> s
0   Linda van der Berg
1  George Pitt-Rivers
dtype: object
```

```python
>>> s.str.partition()
          0      1      2
0  Linda  van der Berg
1  George  Pitt-Rivers
```

To partition by the last space instead of the first one:

```python
>>> s.str.rpartition()
          0      1      2
0  Linda  van der  Berg
1  George    Pitt  - Rivers
```

To partition by something different than a space:

```python
>>> s.str.partition('-')
          0      1      2
0  Linda  van der  Berg
1  George  Pitt - Rivers
```

To return a Series containing tuples instead of a DataFrame:

```python
>>> s.str.partition('-', expand=False)
0  (Linda van der Berg, , )
1  (George Pitt, -, Rivers)
dtype: object
```

Also available on indices:

```python
>>> idx = pd.Index(['X 123', 'Y 999'])
>>> idx
Index(['X 123', 'Y 999'], dtype='object')
```

Which will create a MultiIndex:

```python
>>> idx.str.partition()
MultiIndex([['X', ' ', '123'],
            ['Y', ' ', '999']],
           )
```

Or an index with tuples with `expand=False`:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> idx.str.partition(expand=False)
Index([('X', ' ', '123'), ('Y', ' ', '999')], dtype='object')
```

**pandas.Series.str.rstrip**

`Series.str.rstrip(to_strip=None)`

Remove trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from right side. Equivalent to `str.rstrip()`.

**Parameters**

- `to_strip` [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

**Returns**

Series or Index of object

**See also:**

- `Series.str.strip` Remove leading and trailing characters in Series/Index.
- `Series.str.lstrip` Remove leading characters in Series/Index.
- `Series.str.rstrip` Remove trailing characters in Series/Index.

**Examples**

```python
>>> s = pd.Series(['1. Ant. ', '2. Bee!
', '3. Cat?\t', np.nan])
>>> s
0  1. Ant.
1   2. Bee!
2   3. Cat?
3     NaN
dtype: object

>>> s.str.strip()
0  1. Ant.
1   2. Bee!
2   3. Cat?
3     NaN
dtype: object

>>> s.str.lstrip('123. ')
0   Ant.
1   Bee!
2   Cat?
3     NaN
dtype: object

>>> s.str.rstrip('!.? \n\t')
0   1. Ant
(continues on next page)```
pandas.Series.str.slice

Series.str.slice\(\text{start}=\text{None}, \text{stop}=\text{None}, \text{step}=\text{None}\)

Slice substrings from each element in the Series or Index.

Parameters

- **start** [int, optional] Start position for slice operation.
- **stop** [int, optional] Stop position for slice operation.
- **step** [int, optional] Step size for slice operation.

Returns

Series or Index of object Series or Index from sliced substring from original string object.

See also:

Series.str.slice_replace Replace a slice with a string.
Series.str.get Return element at position. Equivalent to Series.str.slice\(\text{start}=i, \text{stop}=i+1\) with \(i\) being the position.

Examples

```python
>>> s = pd.Series(["koala", "dog", "chameleon"])
>>> s
0    koala
1      dog
2  chameleon
dtype: object

>>> s.str.slice(start=1)
0    oala
1     og
2  hameleon
dtype: object

>>> s.str.slice(start=-1)
0    a
1    g
```
2    n
dtype: object

>>> s.str.slice(stop=2)
0    ko
1    do
2    ch
dtype: object

>>> s.str.slice(step=2)
0    kaa
1    dg
2    caeen
dtype: object

>>> s.str.slice(start=0, stop=5, step=3)
0    kl
1    d
2    cm
dtype: object

Equivalent behaviour to:

>>> s.str[0:5:3]
0    kl
1    d
2    cm
dtype: object

**pandas.Series.str.slice_replace**

Series.str.slice_replace(start=None, stop=None, repl=None)

Replace a positional slice of a string with another value.

**Parameters**

- **start** [int, optional] Left index position to use for the slice. If not specified (None), the slice is unbounded on the left, i.e. slice from the start of the string.

- **stop** [int, optional] Right index position to use for the slice. If not specified (None), the slice is unbounded on the right, i.e. slice until the end of the string.

- **repl** [str, optional] String for replacement. If not specified (None), the sliced region is replaced with an empty string.

**Returns**

- Series or Index Same type as the original object.

**See also:**

- Series.str.slice Just slicing without replacement.
Examples

```python
>>> s = pd.Series(['a', 'ab', 'abc', 'abdc', 'abcde'])
>>> s
0    a
1    ab
2   abc
3  abdc
4 abcde
dtype: object
```

Specify just `start`, meaning replace `start` until the end of the string with `repl`.

```python
>>> s.str.slice_replace(1, repl='X')
0   aX
1  aX
2  aX
3  aX
4  aX
dtype: object
```

Specify just `stop`, meaning the start of the string to `stop` is replaced with `repl`, and the rest of the string is included.

```python
>>> s.str.slice_replace(stop=2, repl='X')
0   Xc
1  Xdc
2   Xc
3  Xdc
4  Xcde
dtype: object
```

Specify `start` and `stop`, meaning the slice from `start` to `stop` is replaced with `repl`. Everything before or after `start` and `stop` is included as is.

```python
>>> s.str.slice_replace(start=1, stop=3, repl='X')
0   aX
1  aX
2  aX
3  aX
4  aXde
dtype: object
```

**pandas.Series.str.split**

`Series.str.split(pat=None, n=- 1, expand=False, *, regex=None)`

Split strings around given separator/delimiter.

Splits the string in the Series/Index from the beginning, at the specified delimiter string.

**Parameters**

- **pat** [str or compiled regex, optional] String or regular expression to split on. If not specified, split on whitespace.
n [int, default -1 (all)] Limit number of splits in output. None, 0 and -1 will be interpreted as return all splits.

expand [bool, default False] Expand the split strings into separate columns.
- If True, return DataFrame/MultiIndex expanding dimensionality.
- If False, return Series/Index, containing lists of strings.

regex [bool, default None] Determines if the passed-in pattern is a regular expression:
- If True, assumes the passed-in pattern is a regular expression
- If False, treats the pattern as a literal string.
- If None and pat length is 1, treats pat as a literal string.
- If None and pat length is not 1, treats pat as a regular expression.
- Cannot be set to False if pat is a compiled regex

New in version 1.4.0.

Returns

Series, Index, DataFrame or MultiIndex Type matches caller unless expand=True (see Notes).

Raises

ValueError
- if regex is False and pat is a compiled regex

See also:

Series.str.split Split strings around given separator/delimiter.
Series.str.rsplit Splits string around given separator/delimiter, starting from the right.
Series.str.join Join lists contained as elements in the Series/Index with passed delimiter.
str.split Standard library version for split.
str.rsplit Standard library version for rsplit.

Notes

The handling of the n keyword depends on the number of found splits:
- If found splits > n, make first n splits only
- If found splits <= n, make all splits
- If for a certain row the number of found splits < n, append None for padding up to n if expand=True

If using expand=True, Series and Index callers return DataFrame and MultiIndex objects, respectively.

Use of regex=False with a pat as a compiled regex will raise an error.

Examples

```
>>> s = pd.Series(
...     [...
...         "this is a regular sentence",
...         "https://docs.python.org/3/tutorial/index.html",
...         np.nan
...     ]
... )
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

>>> s
0
this is a regular sentence
1
https://docs.python.org/3/tutorial/index.html
2
NaN
dtype: object
In the default setting, the string is split by whitespace.
>>> s.str.split()
0
[this, is, a, regular, sentence]
1
[https://docs.python.org/3/tutorial/index.html]
2
NaN
dtype: object
Without the n parameter, the outputs of rsplit and split are identical.
>>> s.str.rsplit()
0
[this, is, a, regular, sentence]
1
[https://docs.python.org/3/tutorial/index.html]
2
NaN
dtype: object
The n parameter can be used to limit the number of splits on the delimiter. The outputs of split and rsplit are
different.
>>> s.str.split(n=2)
0
[this, is, a regular sentence]
1
[https://docs.python.org/3/tutorial/index.html]
2
NaN
dtype: object
>>> s.str.rsplit(n=2)
0
[this is a, regular, sentence]
1
[https://docs.python.org/3/tutorial/index.html]
2
NaN
dtype: object
The pat parameter can be used to split by other characters.
>>> s.str.split(pat="/")
0
[this is a regular sentence]
1
[https:, , docs.python.org, 3, tutorial, index...
2
NaN
dtype: object
When using expand=True, the split elements will expand out into separate columns. If NaN is present, it is
propagated throughout the columns during the split.
>>> s.str.split(expand=True)
0
1
2

0
this
https://docs.python.org/3/tutorial/index.html
NaN

3.3. Series

1
is
None
NaN

2
a
None
NaN

3
regular
None
NaN

4
sentence
None
NaN

1523


For slightly more complex use cases like splitting the html document name from a url, a combination of parameter settings can be used.

```python
s = pd.Series(["this is a regular sentence", "https://docs.python.org/3/tutorial index.html", NaN, NaN])
```

Remember to escape special characters when explicitly using regular expressions.

```python
s = pd.Series(["foo and bar plus baz"]) s.str.split(r"and|plus", expand=True)
```

Regular expressions can be used to handle urls or filenames. When `pat` is a string and `regex=None` (the default), the given `pat` is compiled as a regex only if `len(pat) != 1`.

```python
s = pd.Series(["foojpgbar.jpg"]) s.str.split(r".", expand=True)
```

When `regex=True`, `pat` is interpreted as a regex

```python
s = pd.Series(["foo.jpg"])
s.str.split(r"\..", regex=True, expand=True)
```

A compiled regex can be passed as `pat`

```python
import re
s.str.split(re.compile(r"\.."), expand=True)
```

When `regex=False`, `pat` is interpreted as the string itself

```python
s.str.split(r"\..", regex=False, expand=True)
```

```
pandas.Series.str.rsplit

Series.str.rsplit(pat=None, n=-1, expand=False)

Split strings around given separator/delimiter.

Splits the string in the Series/Index from the end, at the specified delimiter string.

Parameters

- **pat** [str or compiled regex, optional] String or regular expression to split on. If not specified, split on whitespace.
- **n** [int, default -1 (all)] Limit number of splits in output. None, 0 and -1 will be interpreted as return all splits.
- **expand** [bool, default False] Expand the split strings into separate columns.
  - If True, return DataFrame/MultiIndex expanding dimensionality.
  - If False, return Series/Index, containing lists of strings.
- **regex** [bool, default None] Determines if the passed-in pattern is a regular expression:
  - If True, assumes the passed-in pattern is a regular expression
  - If False, treats the pattern as a literal string.
  - If None and pat length is 1, treats pat as a literal string.
  - If None and pat length is not 1, treats pat as a regular expression.
  - Cannot be set to False if pat is a compiled regex

New in version 1.4.0.

Returns

- **Series, Index, DataFrame or MultiIndex** Type matches caller unless expand=True (see Notes).

Raises

- **ValueError**
  - if regex is False and pat is a compiled regex

See also:

- **Series.str.split** Split strings around given separator/delimiter.
- **Series.str.rsplit** Splits string around given separator/delimiter, starting from the right.
- **Series.str.join** Join lists contained as elements in the Series/Index with passed delimiter.
- **str.split** Standard library version for split.
- **str.rsplit** Standard library version for rsplit.

Notes

The handling of the n keyword depends on the number of found splits:

- If found splits > n, make first n splits only
- If found splits <= n, make all splits

- If for a certain row the number of found splits < n, append None for padding up to n if expand=True

If using expand=True, Series and Index callers return DataFrame and MultiIndex objects, respectively.

Use of regex=False with a pat as a compiled regex will raise an error.
Examples

```python
>>> s = pd.Series(
...    ["this is a regular sentence",
...     "https://docs.python.org/3/tutorial/index.html",
...     np.nan
...    ]
...)
```

```python
>>> s
0     this is a regular sentence
1  https://docs.python.org/3/tutorial/index.html
2      NaN
dtype: object
```

In the default setting, the string is split by whitespace.

```python
>>> s.str.split()
0   [this, is, a, regular, sentence]
1  [https://docs.python.org/3/tutorial/index.html]
2      NaN
dtype: object
```

Without the `n` parameter, the outputs of `rsplit` and `split` are identical.

```python
>>> s.str.rsplit()
0   [this, is, a, regular, sentence]
1  [https://docs.python.org/3/tutorial/index.html]
2      NaN
dtype: object
```

The `n` parameter can be used to limit the number of splits on the delimiter. The outputs of `split` and `rsplit` are different.

```python
>>> s.str.split(n=2)
0   [this, is, a regular sentence]
1  [https://docs.python.org/3/tutorial/index.html]
2      NaN
dtype: object
```

```python
>>> s.str.rsplit(n=2)
0      [this is a, regular, sentence]
1  [https://docs.python.org/3/tutorial/index.html]
2      NaN
dtype: object
```

The `pat` parameter can be used to split by other characters.

```python
>>> s.str.split(pat=":")
0     [this is a regular sentence]
1  [https:, , docs.python.org, 3, tutorial, index...]
2      NaN
dtype: object
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

When using expand=True, the split elements will expand out into separate columns. If NaN is present, it is
propagated throughout the columns during the split.
>>> s.str.split(expand=True)
0
1
2

0
this
https://docs.python.org/3/tutorial/index.html
NaN

1
is
None
NaN

2
a
None
NaN

3
regular
None
NaN

4
sentence
None
NaN

For slightly more complex use cases like splitting the html document name from a url, a combination of parameter
settings can be used.
>>> s.str.rsplit("/", n=1, expand=True)
0
1
0
this is a regular sentence
None
1 https://docs.python.org/3/tutorial index.html
2
NaN
NaN
Remember to escape special characters when explicitly using regular expressions.
>>> s = pd.Series(["foo and bar plus baz"])
>>> s.str.split(r"and|plus", expand=True)
0
1
2
0 foo bar baz
Regular expressions can be used to handle urls or file names. When pat is a string and regex=None (the default),
the given pat is compiled as a regex only if len(pat) != 1.
>>> s = pd.Series(['foojpgbar.jpg'])
>>> s.str.split(r".", expand=True)
0
1
0 foojpgbar jpg
>>> s.str.split(r"\.jpg", expand=True)
0 1
0 foojpgbar
When regex=True, pat is interpreted as a regex
>>> s.str.split(r"\.jpg", regex=True, expand=True)
0 1
0 foojpgbar
A compiled regex can be passed as pat
>>> import re
>>> s.str.split(re.compile(r"\.jpg"), expand=True)
0 1
0 foojpgbar
When regex=False, pat is interpreted as the string itself
>>> s.str.split(r"\.jpg", regex=False, expand=True)
0
0 foojpgbar.jpg

3.3. Series

1527


pandas.Series.str.startswith

Series.str.startswith(*pat*, *na=None*)

Test if the start of each string element matches a pattern.

Equivalent to str.startswith().

Parameters

- **pat** [str] Character sequence. Regular expressions are not accepted.
- **na** [object, default NaN] Object shown if element tested is not a string. The default depends on dtype of the array. For object-dtype, *numpy.nan* is used. For StringDtype, *pandas.NA* is used.

Returns

Series or Index of bool  A Series of booleans indicating whether the given pattern matches the start of each string element.

See also:

- **str.startswith**  Python standard library string method.
- **Series.str.endswith**  Same as startswith, but tests the end of string.
- **Series.str.contains**  Tests if string element contains a pattern.

Examples

```python
>>> s = pd.Series(['bat', 'Bear', 'cat', np.nan])
>>> s
0    bat
1   Bear
2     cat
3  NaN
dtype: object

>>> s.str.startswith('b')
0    True
1   False
2   False
3   False
dtype: bool
```

Specifying *na* to be *False* instead of *NaN*.

```python
>>> s.str.startswith('b', na=False)
0   False
1   False
2   False
3   False
dtype: bool
```
**pandas.Series.str.strip**

Series.str.strip(to_strip=None)

Remove leading and trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left and right sides. Equivalent to str.strip().

**Parameters**

- **to_strip** [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

**Returns**

Series or Index of object

See also:

- **Series.str.strip** Remove leading and trailing characters in Series/Index.
- **Series.str.lstrip** Remove leading characters in Series/Index.
- **Series.str.rstrip** Remove trailing characters in Series/Index.

**Examples**

```python
>>> s = pd.Series(['1. Ant.', '2. Bee!
', '3. Cat?	', np.nan])
>>> s
0  1. Ant.
1  2. Bee!
2  3. Cat?
3   NaN
dtype: object

>>> s.str.strip()
0  1. Ant.
1  2. Bee!
2  3. Cat?
3   NaN
dtype: object

>>> s.str.lstrip('123. ')
0   Ant.
1  Bee!
2  Cat?
3   NaN
dtype: object

>>> s.str.rstrip('.!? \n\t')
0   1. Ant
1   2. Bee
2   3. Cat
3   NaN
dtype: object
```
>>> s.str.strip('123.!? \n\t')
0    Ant
1    Bee
2    Cat
3   NaN
dtype: object

pandas.Series.str.swapcase

Series.str.swapcase()

Convert strings in the Series/Index to be swapcased.

Equivalent to str.swapcase().

Returns

Series or Index of object

See also:

Series.str.lower  Converts all characters to lowercase.
Series.str.upper  Converts all characters to uppercase.
Series.str.title  Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize  Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase  Converts uppercase to lowercase and lowercase to uppercase.
Series.str.casefold  Removes all case distinctions in the string.

Examples

>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])

>>> s
0     lower
1    CAPITALS
2  this is a sentence
3    SwApCaSe
dtype: object

>>> s.str.lower()
0     lower
1    capitals
2  this is a sentence
3    swapcase
dtype: object

>>> s.str.upper()
0      LOWER
1    CAPITALS
2 THIS IS A SENTENCE
3    SWAPCASE
dtype: object
pandas.Series.str.title

Series.str.title()
Convert strings in the Series/Index to titlecase.
Equivalent to \texttt{str.title()}.  
\textbf{Returns}
Series or Index of object

\textbf{See also:}
\begin{itemize}
\item \texttt{Series.str.lower} Converts all characters to lowercase.
\item \texttt{Series.str.upper} Converts all characters to uppercase.
\item \texttt{Series.str.title} Converts first character of each word to uppercase and remaining to lowercase.
\item \texttt{Series.str.capitalize} Converts first character to uppercase and remaining to lowercase.
\item \texttt{Series.str.swapcase} Converts uppercase to lowercase and lowercase to uppercase.
\item \texttt{Series.str.casefold} Removes all case distinctions in the string.
\end{itemize}

\textbf{Examples}

```python
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0    lower
1  CAPITALS
2  this is a sentence
3    SwApCaSe
dtype: object
```
```python
>>> s.str.lower()
0    lower
1   capitals
2  this is a sentence
3  swapcase
dtype: object

>>> s.str.upper()
0    LOWER
1   CAPITALS
2  THIS IS A SENTENCE
3   SWAPCASE
dtype: object

>>> s.str.title()
0       Lower
1      Capitals
2    This Is A Sentence
3   Swapcase
dtype: object

>>> s.str.capitalize()
0       Lower
1      Capitals
2  This is a sentence
3   Swapcase
dtype: object

>>> s.str.swapcase()
0      LOWER
1    capitals
2  THIS IS A SENTENCE
3  sWaPcAsE
dtype: object
```

**pandas.Series.str.translate**

Series.str.translate(table)

Map all characters in the string through the given mapping table.

Equivalent to standard `str.translate()`.

**Parameters**

- `table` [dict] Table is a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. `str.maketrans()` is a helper function for making translation tables.

**Returns**

- Series or Index
pandas.Series.str.upper

Series.str.upper()

Convert strings in the Series/Index to uppercase.

Equivalent to str.upper().

Returns

Series or Index of object

See also:

Series.str.lower Converts all characters to lowercase.
Series.str.upper Converts all characters to uppercase.
Series.str.title Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.
Series.str.casefold Removes all case distinctions in the string.

Examples

```python
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0    lower
1  CAPITALS
2    this is a sentence
3     SwApCaSe
dtype: object

>>> s.str.lower()
0    lower
1  capitals
2    this is a sentence
3    swapcase
dtype: object

>>> s.str.upper()
0  LOWER
1  CAPITALS
2  THIS IS A SENTENCE
3  SWAPCASE
dtype: object

>>> s.str.title()
0    Lower
1    Capitals
2    This Is A Sentence
3    Swapcase
dtype: object

>>> s.str.capitalize()
0    Lower
1    Capitals
```

(continues on next page)
This is a sentence

```python
s.str.swapcase()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOWER</td>
</tr>
<tr>
<td>1</td>
<td>capitals</td>
</tr>
<tr>
<td>2</td>
<td>THIS IS A SENTENCE</td>
</tr>
<tr>
<td>3</td>
<td>sWaPcAsE</td>
</tr>
</tbody>
</table>

**pandas.Series.str.wrap**

`Series.str.wrap(width, **kwargs)`

Wrap strings in Series/Index at specified line width.

This method has the same keyword parameters and defaults as `textwrap.TextWrapper`.

**Parameters**

- `width` [int] Maximum line width.
- `expand_tabs` [bool, optional] If True, tab characters will be expanded to spaces (default: True).
- `replace_whitespace` [bool, optional] If True, each whitespace character (as defined by `string.whitespace`) remaining after tab expansion will be replaced by a single space (default: True).
- `drop_whitespace` [bool, optional] If True, whitespace that, after wrapping, happens to end up at the beginning or end of a line is dropped (default: True).
- `break_long_words` [bool, optional] If True, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width (default: True).
- `break_on_hyphens` [bool, optional] If True, wrapping will occur preferably on whitespace and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly inseparable words (default: True).

**Returns**

Series or Index

**Notes**

Internally, this method uses a `textwrap.TextWrapper` instance with default settings. To achieve behavior matching R’s stringr library `str_wrap` function, use the arguments:

- `expand_tabs = False`
- `replace_whitespace = True`
- `drop_whitespace = True`
- `break_long_words = False`
- `break_on_hyphens = False`
Examples

```python
>>> s = pd.Series(['line to be wrapped', 'another line to be wrapped'])
>>> s.str.wrap(12)
0   line to be\n     wrapped
1  another line\n    to be\n     wrapped
dtype: object
```

pandas.Series.str.zfill

Series.str.zfill(width)

Pad strings in the Series/Index by prepending ‘0’ characters.

Strings in the Series/Index are padded with ‘0’ characters on the left of the string to reach a total string length `width`. Strings in the Series/Index with length greater or equal to `width` are unchanged.

Parameters

- `width` [int] Minimum length of resulting string; strings with length less than `width` be prepended with ‘0’ characters.

Returns

Series/Index of objects.

See also:

- `Series.str.rjust` Fills the left side of strings with an arbitrary character.
- `Series.str.ljust` Fills the right side of strings with an arbitrary character.
- `Series.str.pad` Fills the specified sides of strings with an arbitrary character.
- `Series.str.center` Fills both sides of strings with an arbitrary character.

Notes

Differs from `str.zfill()` which has special handling for ‘+’/-’ in the string.

Examples

```python
>>> s = pd.Series(['-1', '1', '1000', 10, np.nan])
>>> s
0   -1
1    1
2   1000
3    10
4  NaN
dtype: object
```

Note that 10 and NaN are not strings, therefore they are converted to NaN. The minus sign in ‘-1’ is treated as a regular character and the zero is added to the left of it (str.zfill() would have moved it to the left). 1000 remains unchanged as it is longer than width.

```python
>>> s.str.zfill(3)
0   0-1
1   001
```
pandas.Series.str.isalnum

Series.str.isalnum()  
Check whether all characters in each string are alphanumeric.  

This is equivalent to running the Python string method str.isalnum() for each element of the Series/Index. If a string has zero characters, False is returned for that check.  

Returns  
Series or Index of bool  
Series or Index of boolean values with the same length as the original Series/Index.

See also:  
Series.str.isalpha  
Check whether all characters are alphabetic.  
Series.str.isnumeric  
Check whether all characters are numeric.  
Series.str.isalnum  
Check whether all characters are alphanumeric.  
Series.str.isdigit  
Check whether all characters are digits.  
Series.str.isdecimal  
Check whether all characters are decimal.  
Series.str.isspace  
Check whether all characters are whitespace.  
Series.str.islower  
Check whether all characters are lowercase.  
Series.str.isupper  
Check whether all characters are uppercase.  
Series.str.istitle  
Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0    True
1    False
2    False
3    False
dtype: bool

>>> s1.str.isnumeric()
0    False
1    False
2    True
3    False
dtype: bool

>>> s1.str.isalnum()
0    True
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

### More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```python
>>> s3.str.isdecimal()
0     True
1     False
2     False
3     False
dtype: bool
```

The `s3.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0     True
1     True
2     False
3     False
dtype: bool
```

The `s3.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0     True
1     True
2     True
3     False
dtype: bool
```

### Checks for Whitespace

```python
>>> s4 = pd.Series([' ', '	
 ', ''])
>>> s4.str.isspace()
0     True
1    False
2    False
3    False
dtype: bool
```
Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```python
>>> s5.str.islower()
0    True
1    False
2    False
3    False
dtype: bool
```

```python
>>> s5.str.isupper()
0    False
1    False
2    True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0    False
1    True
2    False
3    False
dtype: bool
```

**pandas.Series.str.isalpha**

`Series.str.isalpha()`  
Check whether all characters in each string are alphabetic.

This is equivalent to running the Python string method `str.isalpha()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

**Returns**

- **Series or Index of bool** Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumeric.
- `Series.str.isdigit` Check whether all characters are digits.
- `Series.str.isdecimal` Check whether all characters are decimal.
Series.str.isspace Check whether all characters are whitespace.
Series.str.islower Check whether all characters are lowercase.
Series.str.isupper Check whether all characters are uppercase.
Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'onel', '1', ''])
```
```
>>> s1.str.isalpha()
0   True
1  False
2  False
3  False
dtype: bool
```
```
>>> s1.str.isnumeric()
0  False
1  False
2   True
3  False
dtype: bool
```
```
>>> s1.str.isalnum()
0   True
1   True
2   True
3  False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
```
```
>>> s2.str.isalnum()
0  False
1  False
2  False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])
```
```
The s3.str.isdecimal method checks for characters used to form numbers in base 10.
```
```
>>> s3.str.isdecimal()
0   True
1   True
```
(continues on next page)
The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0    True
1    True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0    True
1    True
2    True
3    False
dtype: bool
```

### Checks for Whitespace

```python
>>> s4 = pd.Series([' ', '	\n', ''])
>>> s4.str.isspace()
0    True
1    True
2    False
dtype: bool
```

### Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```python
>>> s5.str.islower()
0    True
1    False
2    False
3    False
dtype: bool
```

```python
>>> s5.str.isupper()
0    False
1    False
2    True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by...
whitespace characters.

```python
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

### pandas.Series.str.isdigit

`Series.str.isdigit()` checks whether all characters in each string are digits.

This is equivalent to running the Python string method `str.isdigit()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

**Returns**

- **Series or Index of bool**
  - Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumeric.
- `Series.str.isdigit` Check whether all characters are digits.
- `Series.str.isdecimal` Check whether all characters are decimal.
- `Series.str.isspace` Check whether all characters are whitespace.
- `Series.str.islower` Check whether all characters are lowercase.
- `Series.str.isupper` Check whether all characters are uppercase.
- `Series.str.istitle` Check whether all characters are titlecase.

### Examples

#### Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```python
>>> s1.str.isalpha()
0    True
1   False
2   False
3   False
dtype: bool
```

```python
>>> s1.str.isnumeric()
0   False
1   False
2     True
3   False
dtype: bool
```
>>> s1.str.isalnum()
0   True
1   True
2   True
3   False
dtype: bool

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0   False
1   False
2   False
dtype: bool

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

>>> s3.str.isdecimal()
0   True
1   False
2   False
3   False
dtype: bool

The s.str.isdigit method is the same as s3.str.isdecimal but also includes special digits, like super-scripted and subscripted digits in unicode.

>>> s3.str.isdigit()
0   True
1   True
2   False
3   False
dtype: bool

The s.str.isnumeric method is the same as s3.str.isdigit but also includes other characters that can represent quantities such as unicode fractions.

>>> s3.str.isnumeric()
0   True
1   True
2   True
3   False
dtype: bool

Checks for Whitespace
```python
>>> s4 = pd.Series([' ', '
', ''])
>>> s4.str.isspace()
0    True
1    True
2   False
dtype: bool
```

Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```python
>>> s5.str.islower()
0   True
1  False
2  False
3  False
dtype: bool
```

```python
>>> s5.str.isupper()
0  False
1  False
2   True
3  False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0  False
1   True
2  False
3  False
dtype: bool
```

**pandas.Series.str.isspace**

`Series.str.isspace()`  
Check whether all characters in each string are whitespace.

This is equivalent to running the Python string method `str.isspace()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

**Returns**

Series or Index of bool  
Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumeric.
- `Series.str.isdigit` Check whether all characters are digits.
Series.str.isdecimal  Check whether all characters are decimal.
Series.str.isspace  Check whether all characters are whitespace.
Series.str.islower  Check whether all characters are lowercase.
Series.str.isupper  Check whether all characters are uppercase.
Series.str.istitle  Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0    True
1    False
2    False
3    False
dtype: bool

>>> s1.str.isnumeric()
0    False
1    False
2    True
3    False
dtype: bool

>>> s1.str.isalnum()
0    True
1    True
2    True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])

>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

```python
>>> s3.str.isdecimal()
0    True
```
The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0   True
1   True
2   False
3   False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0   True
1   True
2   True
3   False
dtype: bool
```

Checks for Whitespace

```python
>>> s4 = pd.Series([' ', '	\n ', ''])
>>> s4.str.isspace()
0   True
1   True
2   False
dtype: bool
```

Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])

>>> s5.str.islower()
0   True
1   False
2   False
3   False
dtype: bool
```

```python
>>> s5.str.isupper()
0   False
1   False
2   True
3   False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of
pandas: powerful Python data analysis toolkit, Release 1.4.2

Each word is capitalized. Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

**pandas.Series.str.islower**

Series.str.islower()

Check whether all characters in each string are lowercase.

This is equivalent to running the Python string method `str.islower()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

**Returns**

Series or Index of bool

Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumeric.
- `Series.str.isdigit` Check whether all characters are digits.
- `Series.str.isdecimal` Check whether all characters are decimal.
- `Series.str.isspace` Check whether all characters are whitespace.
- `Series.str.islower` Check whether all characters are lowercase.
- `Series.str.isupper` Check whether all characters are uppercase.
- `Series.str.istitle` Check whether all characters are titlecase.

**Examples**

**Checks for Alphabetic and Numeric Characters**

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0   True
1  False
2  False
3  False
dtype: bool

>>> s1.str.isnumeric()
0   False
1  False
2   True
3  False
dtype: bool
```
Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```python
>>> s3.str.isdecimal()
0   True
1   False
2   False
3   False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0   True
1   True
2   False
3   False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0   True
1   True
2   True
3   False
dtype: bool
```

Checks for Whitespace
```python
>>> s4 = pd.Series([' ', '	\r\n', ''])
>>> s4.str.isspace()
0    True  
1    True  
2   False  
dtype: bool

Checks for Character Case

```n```n
```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
>>> s5.str.islower()
0   True
1  False
2  False
3  False
dtype: bool

```n```n
```python
>>> s5.str.isupper()
0   False
1  False
2   True
3  False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0  False
1   True
2  False
3  False
dtype: bool
```

**pandas.Series.str.isupper**

`Series.str.isupper()`  
Check whether all characters in each string are uppercase.  
This is equivalent to running the Python string method `str.isupper()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.  
**Returns**  
`Series or Index of bool`  
Series or Index of boolean values with the same length as the original Series/Index.  
**See also:**  
- `Series.str.isalpha` Check whether all characters are alphabetic.  
- `Series.str.isnumeric` Check whether all characters are numeric.  
- `Series.str.isalnum` Check whether all characters are alphanumeric.  
- `Series.str.isdigit` Check whether all characters are digits.
Series.str.isdecimal Check whether all characters are decimal.
Series.str.isspace Check whether all characters are whitespace.
Series.str.islower Check whether all characters are lowercase.
Series.str.isupper Check whether all characters are uppercase.
Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0    True
1   False
2   False
3   False
dtype: bool

>>> s1.str.isnumeric()
0   False
1   False
2    True
3   False
dtype: bool

>>> s1.str.isalnum()
0    True
1    True
2    True
3   False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])

>>> s2.str.isalnum()
0   False
1   False
2   False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

```python
>>> s3.str.isdecimal()
0    True
```

(continues on next page)
The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0    True
1    True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0    True
1    True
2    True
3    False
dtype: bool
```

Checks for Whitespace

```python
>>> s4 = pd.Series([' ', '	
', ''])
>>> s4.str.isspace()
0    True
1    True
2    False
dtype: bool
```

Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
>>> s5.str.islower()
0    True
1    False
2    False
3    False
dtype: bool
```

```python
>>> s5.str.isupper()
0    False
1    False
2    True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of
each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0   False
1    True
2   False
3   False
dtype: bool
```

**pandas.Series.str.istitle**

Series.str.istitle()

Check whether all characters in each string are titlecase.

This is equivalent to running the Python string method `str.istitle()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

**Returns**

`Series` or `Index` of `bool` Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumeric.
- `Series.str.isdigit` Check whether all characters are digits.
- `Series.str.isdecimal` Check whether all characters are decimal.
- `Series.str.isspace` Check whether all characters are whitespace.
- `Series.str.islower` Check whether all characters are lowercase.
- `Series.str.isupper` Check whether all characters are uppercase.
- `Series.str.istitle` Check whether all characters are titlecase.

**Examples**

**Checks for Alphabetic and Numeric Characters**

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0   True
1  False
2  False
3  False
dtype: bool

>>> s1.str.isnumeric()
0   False
1  False
2   True
3  False
dtype: bool
```
>>> s1.str.isalnum()
0    True
1    True
2    True
3    False
dtype: bool

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

>>> s3.str.isdecimal()
0    True
1    False
2    False
3    False
dtype: bool

The s.str.isdigit method is the same as s3.str.isdecimal but also includes special digits, like super-scripted and subscripted digits in unicode.

>>> s3.str.isdigit()
0    True
1    True
2    False
3    False
dtype: bool

The s.str.isnumeric method is the same as s3.str.isdigit but also includes other characters that can represent quantities such as unicode fractions.

>>> s3.str.isnumeric()
0    True
1    True
2    True
3    False
dtype: bool

Checks for Whitespace
```python
>>> s4 = pd.Series([' ', '	
', ''])
>>> s4.str.isspace()
0    True
1    True
2   False
dtype: bool
```

Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])

>>> s5.str.islower()
0    True
1   False
2   False
3   False
dtype: bool

>>> s5.str.isupper()
0   False
1   False
2    True
3   False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0   False
1    True
2   False
3   False
dtype: bool
```

**pandas.Series.str.isnumeric**

Series.str.isnumeric()  
Check whether all characters in each string are numeric.  

This is equivalent to running the Python string method `str.isnumeric()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.  

**Returns**  
Series or Index of bool  
Series or Index of boolean values with the same length as the original Series/Index.  

**See also:**  
Series.str.isalpha  Check whether all characters are alphabetic.  
Series.str.isnumeric  Check whether all characters are numeric.  
Series.str.isalnum  Check whether all characters are alphanumeric.  
Series.str.isdigit  Check whether all characters are digits.
**Series.str.isdecimal** Check whether all characters are decimal.
**Series.str.isspace** Check whether all characters are whitespace.
**Series.str.islower** Check whether all characters are lowercase.
**Series.str.isupper** Check whether all characters are uppercase.
**Series.str.istitle** Check whether all characters are titlecase.

### Examples

Checks for Alphabetic and Numeric Characters

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0   True
1   False
2   False
3   False
dtype: bool

>>> s1.str.isnumeric()
0   False
1   False
2   True
3   False
dtype: bool

>>> s1.str.isalnum()
0   True
1   True
2   True
3   False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```python
>>> s2 = pd.Series(['A B', '1.5', '3,000'])

>>> s2.str.isalnum()
0   False
1   False
2   False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```python
>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

```python
>>> s3.str.isdecimal()
0   True
```
(continues on next page)
The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```python
>>> s3.str.isdigit()
0    True
1    True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```python
>>> s3.str.isnumeric()
0    True
1    True
2    True
3    False
dtype: bool
```

### Checks for Whitespace

```python
>>> s4 = pd.Series([' ', '	
', ''])
>>> s4.str.isspace()
0    True
1    True
2    False
dtype: bool
```

### Checks for Character Case

```python
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
>>> s5.str.islower()
0    True
1    False
2    False
3    False
dtype: bool
```

```python
>>> s5.str.isupper()
0    False
1    False
2    True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of...
each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```python
>>> s5.str.istitle()
0   False
1    True
2   False
3   False
dtype: bool
```

### pandas.Series.str.isdecimal

**Series.str.isdecimal()**

Check whether all characters in each string are decimal.

This is equivalent to running the Python string method `str.isdecimal()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

**Returns**

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

**See also:**

- `Series.str.isalpha` Check whether all characters are alphabetic.
- `Series.str.isnumeric` Check whether all characters are numeric.
- `Series.str.isalnum` Check whether all characters are alphanumerical.
- `Series.str.isdigit` Check whether all characters are digits.
- `Series.str.isdecimal` Check whether all characters are decimal.
- `Series.str.isspace` Check whether all characters are whitespace.
- `Series.str.islower` Check whether all characters are lowercase.
- `Series.str.isupper` Check whether all characters are uppercase.
- `Series.str.istitle` Check whether all characters are titlecase.

### Examples

**Checks for Alphabetic and Numeric Characters**

```python
>>> s1 = pd.Series(['one', 'one1', '1', ''])

>>> s1.str.isalpha()
0   True
1  False
2  False
3  False
dtype: bool

>>> s1.str.isnumeric()
0   False
1  False
2   True
3  False
dtype: bool
```
>>> s1.str.isalnum()
0   True
1   True
2   True
3   False
dtype: bool

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0   False
1   False
2   False
dtype: bool

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

>>> s3 = pd.Series(['23', '3', '', ''])

The s3.str.isdecimal method checks for characters used to form numbers in base 10.

>>> s3.str.isdecimal()
0   True
1   False
2   False
3   False
dtype: bool

The s.str.isdigit method is the same as s3.str.isdecimal but also includes special digits, like superscripted and subscripted digits in unicode.

>>> s3.str.isdigit()
0   True
1   True
2   False
3   False
dtype: bool

The s.str.isnumeric method is the same as s3.str.isdigit but also includes other characters that can represent quantities such as unicode fractions.

>>> s3.str.isnumeric()
0   True
1   True
2   True
3   False
dtype: bool

Checks for Whitespace
>>> s4 = pd.Series([' ', '	
', ''])
>>> s4.str.isspace()
0    True
1    True
2   False
dtype: bool

Checks for Character Case

>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])

>>> s5.str.islower()
0    True
1   False
2   False
3   False
dtype: bool

>>> s5.str.isupper()
0   False
1   False
2    True
3   False
dtype: bool

The \texttt{s5.str.istitle} method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

>>> s5.str.istitle()
0   False
1    True
2   False
3   False
dtype: bool

\texttt{pandas.Series.str.get_dummies}

\texttt{Series.str.get_dummies(sep='')}  
\quad \text{Return DataFrame of dummy/indicator variables for Series.}

\text{Each string in Series is split by \texttt{sep} and returned as a DataFrame of dummy/indicator variables.}

\textbf{Parameters}

\begin{itemize}
\item \texttt{sep}  [str, default “\textquoteleft\textquoteleft"] String to split on.
\end{itemize}

\textbf{Returns}

\texttt{DataFrame}  Dummy variables corresponding to values of the Series.

\textbf{See also:}

\texttt{get_dummies}  Convert categorical variable into dummy/indicator variables.
Examples

```python
>>> pd.Series(['a|b', 'a', 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  1  0  0
2  1  0  1
```

```python
>>> pd.Series(['a|b', np.nan, 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  0  0  0
2  1  0  1
```

Categorical accessor

Categorical-dtype specific methods and attributes are available under the `Series.cat` accessor.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.cat.categories</code></td>
<td>The categories of this categorical.</td>
</tr>
<tr>
<td><code>Series.cat.ordered</code></td>
<td>Whether the categories have an ordered relationship.</td>
</tr>
<tr>
<td><code>Series.cat.codes</code></td>
<td>Return Series of codes as well as the index.</td>
</tr>
</tbody>
</table>

**pandas.Series.cat.categories**

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to `categories` is an inplace operation!

**Raises**

- `ValueError` If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

**See also:**

- `rename_categories` Rename categories.
- `reorder_categories` Reorder categories.
- `add_categories` Add new categories.
- `remove_categories` Remove the specified categories.
- `remove_unused_categories` Remove categories which are not used.
- `set_categories` Set the categories to the specified ones.
**pandas.Series.cat.ordered**

Series.cat.ordered

Whether the categories have an ordered relationship.

**pandas.Series.cat.codes**

Series.cat.codes

Return Series of codes as well as the index.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.cat.rename_categories(*args, **kwargs)</td>
<td>Rename categories.</td>
</tr>
<tr>
<td>Series.cat.reorder_categories(*args, **kwargs)</td>
<td>Reorder categories as specified in new_categories.</td>
</tr>
<tr>
<td>Series.cat.add_categories(*args, **kwargs)</td>
<td>Add new categories.</td>
</tr>
<tr>
<td>Series.cat.remove_categories(*args, **kwargs)</td>
<td>Remove the specified categories.</td>
</tr>
<tr>
<td>Series.cat.remove_unused_categories(*args,...)</td>
<td>Remove categories which are not used.</td>
</tr>
<tr>
<td>Series.cat.set_categories(*args, **kwargs)</td>
<td>Set the categories to the specified new_categories.</td>
</tr>
<tr>
<td>Series.cat.as_ordered(*args, **kwargs)</td>
<td>Set the Categorical to be ordered.</td>
</tr>
<tr>
<td>Series.cat.as_unordered(*args, **kwargs)</td>
<td>Set the Categorical to be unordered.</td>
</tr>
</tbody>
</table>

**pandas.Series.cat.rename_categories**

Series.cat.rename_categories(*args, **kwargs)

Rename categories.

**Parameters**

- **new_categories** [list-like, dict-like or callable] New categories which will replace old categories.
  - list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
  - dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.
  - callable: a callable that is called on all items in the old categories and whose return values comprise the new categories.

- **inplace** [bool, default False] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Deprecateped since version 1.3.0.

**Returns**

- **cat** [Categorical or None] Categorical with removed categories or None if inplace=True.

**Raises**

- **ValueError** If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories.

**See also:**

- reorder_categories Reorder categories.
- add_categories Add new categories.
remove_categories  Remove the specified categories.
remove_unused_categories  Remove categories which are not used.
set_categories  Set the categories to the specified ones.

Examples

```python
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like new_categories, extra keys are ignored and categories not in the dictionary are passed through

```python
>>> c.rename_categories({'a': 'A', 'c': 'C'})
['A', 'A', 'b']
Categories (2, object): ['A', 'b']
```

You may also provide a callable to create the new categories

```python
>>> c.rename_categories(lambda x: x.upper())
['A', 'A', 'B']
Categories (2, object): ['A', 'B']
```

pandas.Series.cat.reorder_categories

Series.cat.reorder_categories(*args, **kwargs)
Reorder categories as specified in new_categories.

new_categories need to include all old categories and no new category items.

Parameters

- **new_categories** [Index-like] The categories in new order.
- **ordered** [bool, optional] Whether or not the categorical is treated as a ordered categorical.
  If not given, do not change the ordered information.
- **inplace** [bool, default False] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Deprecated since version 1.3.0.

Returns

- **cat** [Categorical or None] Categorical with removed categories or None if inplace=True.

Raises

- **ValueError** If the new categories do not contain all old category items or any new ones

See also:

- rename_categories  Rename categories.
- add_categories  Add new categories.
- remove_categories  Remove the specified categories.
- remove_unused_categories  Remove categories which are not used.
- set_categories  Set the categories to the specified ones.
pandas.Series.cat.add_categories

Series.cat.add_categories(*args, **kwargs)

Add new categories.

Parameters

new_categories [category or list-like of category] The new categories to be included.

inplace [bool, default False] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical or None] Categorical with new categories added or None if inplace=True.

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories Rename categories.
reorder_categories Reorder categories.
remove_categories Remove the specified categories.
remove_unused_categories Remove categories which are not used.
set_categories Set the categories to the specified ones.

Examples

>>> c = pd.Categorical(['c', 'b', 'c'])

>>> c
['c', 'b', 'c']
Categories (2, object): ['b', 'c']

>>> c.add_categories(['d', 'a'])
['c', 'b', 'c', 'd', 'a']
Categories (4, object): ['b', 'c', 'd', 'a']

pandas.Series.cat.remove_categories

Series.cat.remove_categories(*args, **kwargs)

Remove the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [bool, default False] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical or None] Categorical with removed categories or None if inplace=True.

Raises

ValueError If the removals do not include old categories or do not validate as categories

See also:

rename_categories Rename categories.
reorder_categories Reorder categories.
remove_categories Remove the specified categories.
remove_unused_categories Remove categories which are not used.
set_categories Set the categories to the specified ones.

Examples

>>> c = pd.Categorical(['c', 'b', 'c'])

>>> c
['c', 'b', 'c']
Categories (2, object): ['b', 'c']

>>> c.remove_categories('b')
['c', 'c']
Categories (1, object): ['c']
Returns

**cat**  [Categorical or None] Categorical with removed categories or None if `inplace=True`.

Raises

**ValueError** If the removals are not contained in the categories

See also:

rename_categories Rename categories.
reorder_categories Reorder categories.
add_categories Add new categories.
remove_unused_categories Remove categories which are not used.
set_categories Set the categories to the specified ones.

Examples

```python
>>> c = pd.Categorical(['a', 'c', 'b', 'c', 'd'])
>>> c
['a', 'c', 'b', 'c', 'd']
Categories (4, object): ['a', 'b', 'c', 'd']

>>> c.remove_categories(['d', 'a'])
[NaN, 'c', 'b', 'c', NaN]
Categories (2, object): ['b', 'c']
```

**pandas.Series.cat.remove_unused_categories**

Series.cat.remove_unused_categories(*args, **kwargs)

Remove categories which are not used.

Parameters

inplace  [bool, default False] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Deprecated since version 1.2.0.

Returns

**cat**  [Categorical or None] Categorical with unused categories dropped or None if inplace=True.

See also:

rename_categories Rename categories.
reorder_categories Reorder categories.
add_categories Add new categories.
remove_categories Remove the specified categories.
set_categories Set the categories to the specified ones.
Examples

```python
>>> c = pd.Categorical(['a', 'c', 'b', 'c', 'd'])
>>> c
['a', 'c', 'b', 'c', 'd']
Categories (4, object): ['a', 'b', 'c', 'd']

>>> c[2] = 'a'
>>> c[4] = 'c'
>>> c
['a', 'c', 'a', 'c', 'c']
Categories (4, object): ['a', 'b', 'c', 'd']

>>> c.remove_unused_categories()
['a', 'c', 'a', 'c']
Categories (2, object): ['a', 'c']
```

**pandas.Series.cat.set_categories**

```python
pandas.Series.cat.set_categories(*args, **kwargs)
```

Set the categories to the specified new_categories.

*new_categories* can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes, which does not considers a S1 string equal to a single char python string.

**Parameters**

- **new_categories** [Index-like] The categories in new order.
- **ordered** [bool, default False] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.
- **rename** [bool, default False] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.
- **inplace** [bool, default False] Whether or not to reorder the categories in-place or return a copy of this categorical with reordered categories.

**Returns**

- Categorical with reordered categories or None if inplace.

**Raises**

- **ValueError** If new_categories does not validate as categories

**See also:**

- `rename_categories` Rename categories.
- `reorder_categories` Reorder categories.
**add_categories** Add new categories.
**remove_categories** Remove the specified categories.
**remove_unused_categories** Remove categories which are not used.

**pandas.Series.cat.as_ordered**

Series.cat.as_ordered(*args, **kwargs)

Set the Categorical to be ordered.

**Parameters**

- **inplace** [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to True.

**Returns**

- **Categorical or None** Ordered Categorical or None if inplace=True.

**pandas.Series.cat.as_unordered**

Series.cat.as_unordered(*args, **kwargs)

Set the Categorical to be unordered.

**Parameters**

- **inplace** [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to False.

**Returns**

- **Categorical or None** Unordered Categorical or None if inplace=True.

**Sparse accessor**

Sparse-dtype specific methods and attributes are provided under the Series.sparse accessor.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.sparse.npoints</td>
<td>The number of non- fill_value points.</td>
</tr>
<tr>
<td>Series.sparse.density</td>
<td>The percent of non- fill_value points, as decimal.</td>
</tr>
<tr>
<td>Series.sparse.fill_value</td>
<td>Elements in data that are fill_value are not stored.</td>
</tr>
<tr>
<td>Series.sparse.sp_values</td>
<td>An ndarray containing the non- fill_value values.</td>
</tr>
</tbody>
</table>

**pandas.Series.sparse.npoints**

Series.sparse.npoints

The number of non- fill_value points.
Examples

```python
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.npoints
3
```

**pandas.Series.sparse.density**

Series.sparse.density

The percent of non-`fill_value` points, as decimal.

```python
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.density
0.6
```

**pandas.Series.sparse.fill_value**

Series.sparse.fill_value

Elements in `data` that are `fill_value` are not stored.
For memory savings, this should be the most common value in the array.

**pandas.Series.sparse.sp_values**

Series.sparse.sp_values

An ndarray containing the non-`fill_value` values.

```python
>>> s = SparseArray([0, 0, 1, 0, 2], fill_value=0)
>>> s.sp_values
array([1, 2])
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.sparse.from_coo(A[, dense_index])</code></td>
<td>Create a Series with sparse values from a scipy.sparse.coo_matrix.</td>
</tr>
<tr>
<td><code>Series.sparse.to_coo([row_levels, ...])</code></td>
<td>Create a scipy.sparse.coo_matrix from a Series with MultiIndex.</td>
</tr>
</tbody>
</table>
**pandas.Series.sparse.from_coo**

*classmethod* Series.sparse.from_coo(A, dense_index=False)*

Create a Series with sparse values from a scipy.sparse.coo_matrix.

**Parameters**

- **A** [scipy.sparse.coo_matrix]
  - *dense_index* [bool, default False] If False (default), the SparseSeries index consists of only the coords of the non-null entries of the original coo_matrix. If True, the SparseSeries index consists of the full sorted (row, col) coordinates of the coo_matrix.

**Returns**

- **s** [Series] A Series with sparse values.

**Examples**

```python
>>> from scipy import sparse

>>> A = sparse.coo_matrix((...([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])), shape=(3, 4))

>>> A
todense()

<3x4 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in COOrdinate format>

>>> A.todense()

matrix([[0., 0., 1., 2.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])

>>> ss = pd.Series.sparse.from_coo(A)

>>> ss
0 2 1.0
3 2.0
1 0 3.0
dtype: Sparse[float64, nan]
```

**pandas.Series.sparse.to_coo**

Series.sparse.to_coo(row_levels=(0,), column_levels=(1,), sort_labels=False)*

Create a scipy.sparse.coo_matrix from a Series with MultiIndex.

Use row_levels and column_levels to determine the row and column coordinates respectively. row_levels and column_levels are the names (labels) or numbers of the levels. {row_levels, column_levels} must be a partition of the MultiIndex level names (or numbers).

**Parameters**

- **row_levels** [tuple/list]
- **column_levels** [tuple/list]
sort_labels [bool, default False] Sort the row and column labels before forming the sparse matrix. When row_levels and/or column_levels refer to a single level, set to True for a faster execution.

Returns

y [scipy.sparse.coo_matrix]
rows [list (row labels)]
columns [list (column labels)]

Examples

```python
>>> s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])
>>> s.index = pd.MultiIndex.from_tuples(
...     [1, 2, "a", 0],
...     [1, 2, "a", 1],
...     [1, 1, "b", 0],
...     [1, 1, "b", 1],
...     [2, 1, "b", 0],
...     [2, 1, "b", 1]
...     ),
...     names=['A', 'B', 'C', 'D'],
... )
>>> s
A B C D
1 2 a 0 3.0
1 NaN
1 b 0 1.0
1 3.0
2 1 b 0 NaN
1 NaN
dtype: float64

>>> ss = s.astype("Sparse")
>>> ss
A B C D
1 2 a 0 3.0
1 NaN
1 b 0 1.0
1 3.0
2 1 b 0 NaN
1 NaN
dtype: Sparse[float64, nan]

>>> A, rows, columns = ss.sparse.to_coo(
...     row_levels=['A', 'B'], column_levels=['C', 'D'], sort_labels=True
... )
>>> A
<3x4 sparse matrix of type 'class 'numpy.float64''
with 3 stored elements in COOrdinate format>
```
Flags

Flags refer to attributes of the pandas object. Properties of the dataset (like the date it was recorded, the URL it was accessed from, etc.) should be stored in `Series.attrs`.

Flags

Flags that apply to pandas objects.

pandas.Flags

```python
class pandas.Flags(obj, *, allows_duplicate_labels)
```

Flags that apply to pandas objects.

New in version 1.2.0.

Parameters

- **obj** [Series or DataFrame] The object these flags are associated with.
- **allows_duplicate_labels** [bool, default True] Whether to allow duplicate labels in this object. By default, duplicate labels are permitted. Setting this to `False` will cause an `errors.DuplicateLabelError` to be raised when `index` (or columns for DataFrame) is not unique, or any subsequent operation on introduces duplicates. See Disallowing Duplicate Labels for more.

Warning: This is an experimental feature. Currently, many methods fail to propagate the `allows_duplicate_labels` value. In future versions it is expected that every method taking or returning one or more DataFrame or Series objects will propagate `allows_duplicate_labels`.

Notes

Attributes can be set in two ways

```python
>>> df = pd.DataFrame()
>>> df.flags
<Flags(allow_duplicates=True)>
>>> df.flags.allows_duplicate_labels = False
>>> df.flags
<Flags(allow_duplicates=False)>
```
>>> df.flags['allows_duplicate_labels'] = True
>>> df.flags
<Flags(allows_duplicate_labels=True)>

Attributes

<table>
<thead>
<tr>
<th>allows_duplicate_labels</th>
<th>Whether this object allows duplicate labels.</th>
</tr>
</thead>
</table>

**pandas.Flags.allows_duplicate_labels**

**property Flags.allows_duplicate_labels**

Whether this object allows duplicate labels.

Setting `allows_duplicate_labels=False` ensures that the index (and columns of a DataFrame) are unique. Most methods that accept and return a Series or DataFrame will propagate the value of `allows_duplicate_labels`.

See *Duplicate Labels* for more.

**See also:**

*DataFrame.attrs* Set global metadata on this object.

*DataFrame.set_flags* Set global flags on this object.

**Examples**

```python
>>> df = pd.DataFrame({'A': [1, 2], 'index': ['a', 'a']})
>>> df.flags.allows_duplicate_labels
True
>>> df.flags.allows_duplicate_labels = False
Traceback (most recent call last):
  ...  
pandas.errors.DuplicateLabelError: Index has duplicates.
  positions
  label a [0, 1]
```

**Metadata**

*Series.attrs* is a dictionary for storing global metadata for this Series.

**Warning:** *Series.attrs* is considered experimental and may change without warning.

| Series.attrs | Dictionary of global attributes of this dataset. |
3.3.14 Plotting

Series.plot is both a callable method and a namespace attribute for specific plotting methods of the form Series.plot.<kind>.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.plot( [kind, ax, figsize, ....])</td>
<td>Series plotting accessor and method</td>
</tr>
<tr>
<td>Series.plot.area(x, y)</td>
<td>Draw a stacked area plot.</td>
</tr>
<tr>
<td>Series.plot.bar(x, y)</td>
<td>Vertical bar plot.</td>
</tr>
<tr>
<td>Series.plot.barh(x, y)</td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td>Series.plot.box([by])</td>
<td>Make a box plot of the DataFrame columns.</td>
</tr>
<tr>
<td>Series.plot.density([bw_method, ind])</td>
<td>Generate Kernel Density Estimate plot using Gaussian kernels.</td>
</tr>
<tr>
<td>Series.plot.hist([by, bins])</td>
<td>Draw one histogram of the DataFrame's columns.</td>
</tr>
<tr>
<td>Series.plot.kde([bw_method, ind])</td>
<td>Generate Kernel Density Estimate plot using Gaussian kernels.</td>
</tr>
<tr>
<td>Series.plot.line(x, y)</td>
<td>Plot Series or DataFrame as lines.</td>
</tr>
<tr>
<td>Series.plot.pie(**kwargs)</td>
<td>Generate a pie plot.</td>
</tr>
</tbody>
</table>

**pandas.Series.plot.area**

Series.plot.area(x=None, y=None, **kwargs)

Draw a stacked area plot.

An area plot displays quantitative data visually. This function wraps the matplotlib area function.

**Parameters**

- x [label or position, optional] Coordinates for the X axis. By default uses the index.
- y [label or position, optional] Column to plot. By default uses all columns.
- stacked [bool, default True] Area plots are stacked by default. Set to False to create a unstacked plot.
- **kwargs Additional keyword arguments are documented in DataFrame.plot().**

**Returns**

- matplotlib.axes.Axes or numpy.ndarray Area plot, or array of area plots if subplots is True.

**See also:**

- DataFrame.plot Make plots of DataFrame using matplotlib / pylab.

**Examples**

Draw an area plot based on basic business metrics:

```python
>>> df = pd.DataFrame({
...     "sales": [3, 2, 3, 9, 10, 6],
...     "signups": [5, 5, 6, 12, 14, 13],
...     "visits": [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
... freq='M'))
>>> ax = df.plot.area()
```
Area plots are stacked by default. To produce an unstacked plot, pass `stacked=False`:

```python
>>> ax = df.plot.area(stacked=False)
```

Draw an area plot for a single column:

```python
>>> ax = df.plot.area(y='sales')
```

Draw with a different x:

```python
>>> df = pd.DataFrame({
...     'sales': [3, 2, 3],
...     'visits': [20, 42, 28],
...     'day': [1, 2, 3],
... })
>>> ax = df.plot.area(x='day')
```
pandas.Series.plot.bar

Series.plot.bar(x=None, y=None, **kwargs)

Vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

Parameters

- x [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.
- y [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.
- color [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:
  - A single color string referred to by name, RGB or RGBA code, for instance ‘red’ or ‘#a98d19’.
  - A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each column recursively. For instance ['green', 'yellow'] each column’s bar will be filled in green or yellow, alternatively. If there is only a single column to be plotted, then only the first color from the color list will be used.
  - A dict of the form {column name: color}, so that each column will be colored accordingly. For example, if your columns are called a and b, then passing {'a': 'green', 'b': 'red'} will color bars for column a in green and bars for column b in red.

New in version 1.1.0.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or np.ndarray of them An ndarray is returned with one matplotlib.axes.Axes per column when subplots=True.

See also:

DataFrame.plot.barch Horizontal bar plot.
DataFrame.plot Make plots of a DataFrame.
matplotlib.pyplot.bar Make a bar plot with matplotlib.

Examples

Basic plot.

```python
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val':[10, 30, 20]})
>>> ax = df.plot.bar(x='lab', y='val', rot=0)
```

Plot a whole dataframe to a bar plot. Each column is assigned a distinct color, and each row is nested in a group along the horizontal axis.
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
                     'lifespan': lifespan}, index=index)
>>> ax = df.plot.bar(rot=0)

Plot stacked bar charts for the DataFrame

>>> ax = df.plot.bar(stacked=True)

Instead of nesting, the figure can be split by column with subplots=True. In this case, a numpy.ndarray of matplotlib.axes.Axes are returned.

>>> axes = df.plot.bar(rot=0, subplots=True)
>>> axes[1].legend(loc=2)

If you don’t like the default colours, you can specify how you’d like each column to be colored.

>>> axes = df.plot.bar(
...     rot=0, subplots=True, color={'speed': "red", "lifespan": "green"}
... )

(continues on next page)
Plot a single column.

```python
>>> ax = df.plot.bar(y='speed', rot=0)
```

Plot only selected categories for the DataFrame.

```python
>>> ax = df.plot.bar(x='lifespan', rot=0)
```

### pandas.Series.plot.barh

**Series.plot.barh**

`Series.plot.barh(x=None, y=None, **kwargs)`

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

- **x** [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.
y  [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

color  [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:

• A single color string referred to by name, RGB or RGBA code, for instance ‘red’ or ‘#a98d19’.

• A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each column recursively. For instance ['green','yellow'] each column’s bar will be filled in green or yellow, alternatively. If there is only a single column to be plotted, then only the first color from the color list will be used.

• A dict of the form {column name: color}, so that each column will be colored accordingly. For example, if your columns are called a and b, then passing {'a': 'green', 'b': 'red'} will color bars for column a in green and bars for column b in red.

New in version 1.1.0.

**kwargs  Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or np.ndarray of them  An ndarray is returned with one matplotlib.axes.Axes per column when subplots=True.

See also:

DataFrame.plot.bar  Vertical bar plot.

DataFrame.plot  Make plots of DataFrame using matplotlib.

matplotlib.axes.Axes.bar  Plot a vertical bar plot using matplotlib.

Examples

Basic example

```python
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> ax = df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant', ...
             'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed, ...
                      'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh()
```

Plot stacked barh charts for the DataFrame

```python
>>> ax = df.plot.barh(stacked=True)
```

We can specify colors for each column
3.3. Series
```python
>>> ax = df.plot.barh(color={"speed": "red", "lifespan": "green"})
```

Plot a column of the DataFrame to a horizontal bar plot

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant', 
...            'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed, 
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(y='speed')
```

Plot DataFrame versus the desired column

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant', 
...            'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed, 
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(x='lifespan')
```
Series.plot.box

Series.plot.box(by=None, **kwargs)

Make a box plot of the DataFrame columns.

A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to 1.5*IQR (IQR = Q3 - Q1) from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

A consideration when using this chart is that the box and the whiskers can overlap, which is very common when plotting small sets of data.

**Parameters**

- **by** [str or sequence] Column in the DataFrame to group by.

  - Changed in version 1.4.0: Previously, by is silently ignore and makes no groupings

- **kwargs** Additional keywords are documented in DataFrame.plot().

**Returns**

- matplotlib.axes.Axes or numpy.ndarray of them

**See also:**

- DataFrame.boxplot Another method to draw a box plot.
- Series.plot.box Draw a box plot from a Series object.
- matplotlib.pyplot.boxplot Draw a box plot in matplotlib.

**Examples**

Draw a box plot from a DataFrame with four columns of randomly generated data.

```python
data = np.random.randn(25, 4)
df = pd.DataFrame(data, columns=list('ABCD'))
ax = df.plot.box()
```

You can also generate groupings if you specify the by parameter (which can take a column name, or a list or tuple of column names):

- Changed in version 1.4.0.

```python
age_list = [8, 10, 12, 14, 72, 74, 76, 78, 20, 25, 30, 35, 60, 85]
df = pd.DataFrame({"gender": list("MMMMMMMMMMMMFFFFF"), "age": age_list})
ax = df.plot.box(column="age", by="gender", figsize=(10, 8))
```
Chapter 3. API reference
**pandas.Series.plot.density**

Series.plot.density(<bw_method>=None, <ind>=None, **<kwargs>**)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters**

- **bw_method** [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See scipy.stats.gaussian_kde for more information.

- **ind** [NumPy array or int, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If ind is a NumPy array, the KDE is evaluated at the points passed. If ind is an integer, ind number of equally spaced points are used.

- **kwargs** Additional keyword arguments are documented in DataFrame.plot().

**Returns**

matplotlib.axes.Axes or numpy.ndarray of them

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

**Examples**

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```python
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```python
>>> ax = s.plot.kde(bw_method=0.3)
```

```python
>>> ax = s.plot.kde(bw_method=3)
```

Finally, the ind parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

For DataFrame, it works in the same way:

```python
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
```

```python
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:
The image shows a line graph with the x-axis labeled from 1.0 to 5.0 and the y-axis labeled as "Density." The graph has a peak at approximately 3.0 on the x-axis and shows a bell-shaped curve, typical of a normal distribution.
>>> ax = df.plot.kde(bw_method=0.3)

Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])

### pandas.Series.plot.hist

`Series.plot.hist(by=None, bins=10, **kwargs)`

Draw one histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`. This is useful when the DataFrame's Series are in a similar scale.

**Parameters**

- **by** [str or sequence, optional] Column in the DataFrame to group by.

  Changed in version 1.4.0: Previously, `by` is silently ignore and makes no groupings

- **bins** [int, default 10] Number of histogram bins to be used.
**kwargs Additional keyword arguments are documented in `DataFrame.plot()`.

Returns

class:`matplotlib.AxesSubplot` Return a histogram plot.

See also:

`DataFrame.hist` Draw histograms per DataFrame’s Series.

`Series.hist` Draw a histogram with Series’ data.

Examples

When we roll a die 6000 times, we expect to get each value around 1000 times. But when we roll two dice and sum the result, the distribution is going to be quite different. A histogram illustrates those distributions.

```python
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns = ['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> ax = df.plot.hist(bins=12, alpha=0.5)
```

A grouped histogram can be generated by providing the parameter `by` (which can be a column name, or a list of column names):
>>> age_list = [8, 10, 12, 14, 72, 74, 76, 78, 20, 25, 30, 35, 60, 85]
>>> df = pd.DataFrame({'gender': list('MMMMMMMMFFFFFF'), 'age': age_list})
>>> ax = df.plot.hist(column=['age'], by='gender', figsize=(10, 8))

pandas.Series.plot.kde

Series.plot.kde(bw_method=None, ind=None, **kwargs)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

Parameters

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See scipy.stats.gaussian_kde for more information.

ind [NumPy array or int, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If ind is a NumPy array, the KDE is evaluated at the points passed. If ind is an integer, ind number of equally spaced points are used.
**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or numpy.ndarray of them

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```python
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```python
>>> ax = s.plot.kde(bw_method=0.3)
```
Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

For DataFrame, it works in the same way:

```python
>>> df = pd.DataFrame({'
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```python
>>> ax = df.plot.kde(bw_method=0.3)
>>> ax = df.plot.kde(bw_method=3)
```

Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:
`pandas.Series.plot.line`

Series.plot.line(x=None, y=None, **kwargs)

Plot Series or DataFrame as lines.

This function is useful to plot lines using DataFrame’s values as coordinates.

**Parameters**

- **x** [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.

- **y** [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

- **color** [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:
  - A single color string referred to by name, RGB or RGBA code, for instance 'red' or '#a98d19'.
  - A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each column recursively. For instance ['green','yellow'] each column’s line will be filled in green or yellow, alternatively. If there is only a
single column to be plotted, then only the first color from the color list will be used.

- A dict of the form {column name: color}, so that each column will be colored accordingly. For example, if your columns are called a and b, then passing {'a': 'green', 'b': 'red'} will color lines for column a in green and lines for column b in red.

New in version 1.1.0.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or np.ndarray of them An ndarray is returned with one matplotlib.axes.Axes per column when subplots=True.

See also:

matplotlib.pyplot.plot Plot y versus x as lines and/or markers.

Examples

```python
>>> s = pd.Series([1, 3, 2])
>>> s.plot.line()
<AxesSubplot:ylabel='Density'>
```
The following example shows the populations for some animals over the years.

```python
>>> df = pd.DataFrame({
...     'pig': [20, 18, 489, 675, 1776],
...     'horse': [4, 25, 281, 600, 1900]
```

An example with subplots, so an array of axes is returned.

```python
>>> axes = df.plot.line(subplots=True)
>>> type(axes)
<class 'numpy.ndarray'>
```

Let's repeat the same example, but specifying colors for each column (in this case, for each animal).

```python
>>> axes = df.plot.line(  
...     subplots=True,  
...     color={'pig': 'pink', 'horse': '#742802'}  
... )
```

The following example shows the relationship between both populations.

```python
>>> lines = df.plot.line(x='pig', y='horse')
```
3.3. Series
pandas.Series.plot.pie

Series.plot.pie(**kwargs)

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps matplotlib.pyplot.pie() for the specified column. If no column reference is passed and subplots=True a pie plot is drawn for each numerical column independently.

Parameters

y [int or label, optional] Label or position of the column to plot. If not provided, subplots=True argument must be passed.

**kwargs Keyword arguments to pass on to DataFrame.plot().

Returns

matplotlib.axes.Axes or np.ndarray of them A NumPy array is returned when subplots is True.

See also:

Series.plot.pie Generate a pie plot for a Series.

DataFrame.plot Make plots of a DataFrame.

Examples

In the example below we have a DataFrame with the information about planet’s mass and radius. We pass the ‘mass’ column to the pie function to get a pie plot.

```python
>>> df = pd.DataFrame({'mass': [0.330, 4.87 , 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                   index=['Mercury', 'Venus', 'Earth'])
>>> plot = df.plot.pie(y='mass', figsize=(5, 5))

>>> plot = df.plot.pie(subplots=True, figsize=(11, 6))
```

Series.hist([by, ax, grid, xlabelsizer, ...]) Draw histogram of the input series using matplotlib.
3.3.15 Serialization / IO / conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.to_pickle(path[, compression, ...])</code></td>
<td>Pickle (serialize) object to file.</td>
</tr>
<tr>
<td><code>Series.to_csv([path_or_buf, sep, na_rep, ...])</code></td>
<td>Write object to a comma-separated values (csv) file.</td>
</tr>
<tr>
<td><code>Series.to_dict([into])</code></td>
<td>Convert Series to {label -&gt; value} dict or dict-like object.</td>
</tr>
<tr>
<td><code>Series.to_excel(excel_writer[, sheet_name, ...])</code></td>
<td>Write object to an Excel sheet.</td>
</tr>
<tr>
<td><code>Series.to_frame([name])</code></td>
<td>Convert Series to DataFrame.</td>
</tr>
<tr>
<td><code>Series.to_hdf(path_or_buf, key[, mode, ...])</code></td>
<td>Write the contained data to an HDF5 file using HDFS-store.</td>
</tr>
<tr>
<td><code>Series.to_json([path_or_buf, orient, ...])</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>Series.to_string([buf, na_rep, ...])</code></td>
<td>Render a string representation of the Series.</td>
</tr>
<tr>
<td><code>Series.to_clipboard([excel, sep])</code></td>
<td>Copy object to the system clipboard.</td>
</tr>
<tr>
<td><code>Series.to_latex([buf, columns, col_space, ...])</code></td>
<td>Render object to a LaTeX tabular, longtable, or nested table.</td>
</tr>
<tr>
<td><code>Series.to_markdown([buf, mode, index, ...])</code></td>
<td>Print Series in Markdown-friendly format.</td>
</tr>
</tbody>
</table>

3.4 DataFrame

3.4.1 Constructor

```python
DataFrame([data, index, columns, dtype, copy])
```

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

pandas.DataFrame

class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters

- **data** [ndarray (structured or homogeneous), Iterable, dict, or DataFrame] Dict can contain Series, arrays, constants, dataclass or list-like objects. If data is a dict, column order follows insertion-order. If a dict contains Series which have an index defined, it is aligned by its index.

  Changed in version 0.25.0: If data is a list of dicts, column order follows insertion-order.

- **index** [Index or array-like] Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided.

- **columns** [Index or array-like] Column labels to use for resulting frame when data does not have them, defaulting to RangeIndex(0, 1, 2, ..., n). If data contains column labels, will perform column selection instead.

- **dtype** [dtype, default None] Data type to force. Only a single dtype is allowed. If None, infer.
copy  [bool or None, default None] Copy data from inputs. For dict data, the default of None behaves like copy=True. For DataFrame or 2d ndarray input, the default of None behaves like copy=False.

Changed in version 1.3.0.

See also:
DataFrame.from_records Constructor from tuples, also record arrays.
DataFrame.from_dict From dicts of Series, arrays, or dicts.
read_csv Read a comma-separated values (csv) file into DataFrame.
read_table Read general delimited file into DataFrame.
read_clipboard Read text from clipboard into DataFrame.

Examples

Constructing DataFrame from a dictionary.

```python
d = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data=d)
df
```

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Notice that the inferred dtype is int64.

```python
df.dtypes
col1  int64
col2  int64
dtype: object
```

To enforce a single dtype:

```python
df = pd.DataFrame(data=d, dtype=np.int8)
df.dtypes
col1  int8
col2  int8
dtype: object
```

Constructing DataFrame from a dictionary including Series:

```python
d = {'col1': [0, 1, 2, 3], 'col2': pd.Series([2, 3], index=[2, 3])}
pd.DataFrame(data=d, index=[0, 1, 2, 3])
```

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Constructing DataFrame from numpy ndarray:

```python
df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
columns=['a', 'b', 'c'])
df2
```

(continues on next page)
Constructing DataFrame from a numpy ndarray that has labeled columns:

```python
>>> data = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)],
                  dtype=[("a", "i4"), ("b", "i4"), ("c", "i4")])
>>> df3 = pd.DataFrame(data, columns=['c', 'a'])
...  
>>> df3
   c a
0 3 1
1 6 4
2 9 7
```

Constructing DataFrame from dataclass:

```python
>>> from dataclasses import make_dataclass
>>> Point = make_dataclass("Point", [("x", int), ("y", int)])
>>> pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
   x  y
0 0 0
1 0 3
2 2 3
```

Attributes

- **at**: Access a single value for a row/column label pair.
- **attrs**: Dictionary of global attributes of this dataset.
- **axes**: Return a list representing the axes of the DataFrame.
- **columns**: The column labels of the DataFrame.
- **dtypes**: Return the dtypes in the DataFrame.
- **empty**: Indicator whether Series/DataFrame is empty.
- **flags**: Get the properties associated with this pandas object.
- **iat**: Access a single value for a row/column pair by integer position.
- **iloc**: Purely integer-location based indexing for selection by position.
- **index**: The index (row labels) of the DataFrame.
- **loc**: Access a group of rows and columns by label(s) or a boolean array.
- **ndim**: Return an int representing the number of axes / array dimensions.
- **shape**: Return a tuple representing the dimensionality of the DataFrame.
- **size**: Return an int representing the number of elements in this object.
- **style**: Returns a Styler object.
- **values**: Return a Numpy representation of the DataFrame.
pandas.DataFrame.at

**property** DataFrame.at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

**Raises**

**KeyError** If ‘label’ does not exist in DataFrame.

**See also:**

- **DataFrame.iat** Access a single value for a row/column pair by integer position.
- **DataFrame.loc** Access a group of rows and columns by label(s).
- **Series.at** Access a single value using a label.

**Examples**

```python
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A  B  C
4  0  2  3
5  0  4  1
6 10 20 30

Get value at specified row/column pair

```python
>>> df.at[4, 'B']
2
```  

Set value at specified row/column pair

```python
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```  

Get value within a Series

```python
>>> df.loc[5].at['B']
4
```
**pandas.DataFrame.attrs**

**property** DataFrame.attrs

Dictionary of global attributes of this dataset.

**Warning:** attrs is experimental and may change without warning.

See also:

*DataFrame.flags* Global flags applying to this object.

**pandas.DataFrame.axes**

**property** DataFrame.axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

**Examples**

```python
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'], dtype='object')]
```

**pandas.DataFrame.columns**

DataFrame.columns

The column labels of the DataFrame.

**pandas.DataFrame.dtypes**

**property** DataFrame.dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result’s index is the original DataFrame’s columns. Columns with mixed types are stored with the object dtype. See the User Guide for more.

**Returns**

* pandas.Series The data type of each column.
Examples

```python
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float       float64
int         int64
datetime    datetime64[ns]
string      object
dtype: object
```

**pandas.DataFrame.empty**

**property** DataFrame.empty

Indicator whether Series/DataFrame is empty.
True if Series/DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

**Returns**

`bool` If Series/DataFrame is empty, return True, if not return False.

**See also:**

*Series.dropna* Return series without null values.

*DataFrame.dropna* Return DataFrame with labels on given axis omitted where (all or any) data are missing.

**Notes**

If Series/DataFrame contains only NaNs, it is still not considered empty. See the example below.

**Examples**

An example of an actual empty DataFrame. Notice the index is empty:

```python
>>> df_empty = pd.DataFrame({'A': []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df = pd.DataFrame({'A': [np.nan]})
>>> df
A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True

>>> ser_empty = pd.Series({'A': []})
>>> ser_empty
A []
dtype: object
>>> ser_empty.empty
False
>>> ser_empty = pd.Series()
>>> ser_empty.empty
True
```

**pandas.DataFrame.flags**

**property DataFrame.flags**
Get the properties associated with this pandas object.

The available flags are

- `Flags.allows_duplicate_labels`

See also:

- `Flags` Flags that apply to pandas objects.
- `DataFrame.attrs` Global metadata applying to this dataset.

**Notes**

“Flags” differ from “metadata”. Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

**Examples**

```python
>>> df = pd.DataFrame({'A': [1, 2]})
>>> df.flags
<Flags(allows_duplicate_labels=True)>
```

Flags can be get or set using .

```python
>>> df.flags.allows_duplicate_labels
True
>>> df.flags.allows_duplicate_labels = False
```

Or by slicing with a key
>>> df.flags["allows_duplicate_labels"]
False
>>> df.flags["allows_duplicate_labels"] = True

**pandas.DataFrame.iat**

**property** DataFrame.iat

Access a single value for a row/column pair by integer position.

Similar to *iloc*, in that both provide integer-based lookups. Use *iat* if you only need to get or set a single value in a DataFrame or Series.

**Raises**

*IndexError* When integer position is out of bounds.

**See also:**

*DataFrame.at* Access a single value for a row/column label pair.

*DataFrame.loc* Access a group of rows and columns by label(s).

*DataFrame.iloc* Access a group of rows and columns by integer position(s).

**Examples**

```python
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30

Get value at specified row/column pair

```python
>>> df.iat[1, 2]
1
```  

Set value at specified row/column pair

```python
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```  

Get value within a series

```python
>>> df.loc[0].iat[1]
2
```
pandas.DataFrameiloc

property DataFrame.iloc

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don’t have a reference to the calling object, but would like to base your selection on some value.

.iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

See more at Selection by Position.

See also:

DataFrame.iat  Fast integer location scalar accessor.

DataFrame.loc  Purely label-location based indexer for selection by label.

Series.iloc  Purely integer-location based indexing for selection by position.

Examples

```python
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
            {'a': 100, 'b': 200, 'c': 300, 'd': 400},
            {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000}]

>>> df = pd.DataFrame(mydict)
>>> df
          a   b   c   d
0       1   2   3   4
1  100  200  300  400
2 1000 2000 3000 4000

Indexing just the rows

With a scalar integer.

```python
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a  1
b  2
c  3
d  4
Name: 0, dtype: int64
```
With a list of integers.

```python
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```python
>>> df.iloc[[0, 1]]
   a  b  c  d
0  1  2  3  4
1 100 200 300 400
```

With a `slice` object.

```python
>>> df.iloc[:3]
   a  b  c  d
0  1  2  3  4
1 100 200 300 400
2 1000 2000 3000 4000
```

With a boolean mask the same length as the index.

```python
>>> df.iloc[[True, False, True]]
   a  b  c  d
0  1  2  3  4
2 1000 2000 3000 4000
```

With a callable, useful in method chains. The `x` passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```python
>>> df.iloc[lambda x: x.index % 2 == 0]
   a  b  c  d
0  1  2  3  4
2 1000 2000 3000 4000
```

**Indexing both axes**

You can mix the indexer types for the index and columns. Use : to select the entire axis.

With scalar integers.

```python
>>> df.iloc[0, 1]
2
```

With lists of integers.

```python
>>> df.iloc[[0, 2], [1, 3]]
   b  d
0  2  4
2 2000 4000
```

With `slice` objects.
>>> df.iloc[1:3, 0:3]
a  b  c
1  100  200  300
2  1000 2000 3000

With a boolean array whose length matches the columns.

>>> df.iloc[:, [True, False, True, False]]
a  c
0  1  3
1  100  300
2  1000 3000

With a callable function that expects the Series or DataFrame.

>>> df.iloc[:, lambda df: [0, 2]]
a  c
0  1  3
1  100  300
2  1000 3000

pandas.DataFrame.index

DataFrame.index
The index (row labels) of the DataFrame.

pandas.DataFrame.loc

property DataFrame.loc
Access a group of rows and columns by label(s) or a boolean array.
.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a': 'f'.

<table>
<thead>
<tr>
<th>Warning:</th>
<th>Note that contrary to usual python slices, both the start and the stop are included</th>
</tr>
</thead>
</table>

- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- An alignable boolean Series. The index of the key will be aligned before masking.
- An alignable Index. The Index of the returned selection will be the input.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

See more at Selection by Label.
Raises

- **KeyError** If any items are not found.
- **IndexingError** If an indexed key is passed and its index is unalignable to the frame index.

See also:

- `DataFrame.at` Access a single value for a row/column label pair.
- `DataFrame.iloc` Access group of rows and columns by integer position(s).
- `DataFrame.xs` Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.
- `Series.loc` Access group of values using labels.

Examples

Getting values

```python
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
     max_speed  shield
cobra      1       2
viper      4       5
sidewinder 7       8
```

Single label. Note this returns the row as a Series.

```python
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using [[]] returns a DataFrame.

```python
>>> df.loc[['viper', 'sidewinder']]
     max_speed  shield
viper         4       5
sidewinder   7       8
```

Single label for row and column

```python
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```python
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```
Boolean list with the same length as the row axis

```python
>>> df.loc[[False, False, True]]
  max_speed  shield
sidewinder   7    8
```

Alignable boolean Series:

```python
>>> df.loc[pd.Series([False, True, False],
index=['viper', 'sidewinder', 'cobra'])]
  max_speed  shield
sidewinder   7    8
```

Index (same behavior as `df.reindex`)

```python
>>> df.loc[pd.Index(['cobra', 'viper'], name='foo')]
  max_speed  shield
foo
  cobra     1    2
  viper     4    5
```

Conditional that returns a boolean Series

```python
>>> df.loc[df['shield'] > 6]
  max_speed  shield
sidewinder   7    8
```

Conditional that returns a boolean Series with column labels specified

```python
>>> df.loc[df['shield'] > 6, ['max_speed']]  
  max_speed
sidewinder   7
```

Callable that returns a boolean Series

```python
>>> df.loc[lambda df: df['shield'] == 8]
  max_speed  shield
sidewinder   7    8
```

**Setting values**

Set value for all items matching the list of labels

```python
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
  max_speed  shield
  cobra     1    2
  viper     4    50
  sidewinder 7    50
```

Set value for an entire row

```python
>>> df.loc['cobra'] = 10
>>> df
  max_speed  shield
  cobra     1    2
  viper     4    50
  sidewinder 7    50
```

(continues on next page)
Set value for an entire column

```python
>>> df.loc[:, 'max_speed'] = 30
>>> df
   max_speed  shield
cobra   30      10
viper   30      50
sidewinder   30      50
```

Set value for rows matching callable condition

```python
>>> df.loc[df['shield'] > 35] = 0
>>> df
   max_speed  shield
cobra   30      10
viper    0       0
sidewinder    0      0
```

**Getting values on a DataFrame with an index that has integer labels**

Another example using integers for the index

```python
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
                      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7  1  2
8  4  5
9  7  8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```python
>>> df.loc[7:9]
   max_speed  shield
7  1  2
8  4  5
9  7  8
```

**Getting values with a MultiIndex**

A number of examples using a DataFrame with a MultiIndex

```python
tuples = [
    ('cobra', 'mark i'), ('cobra', 'mark ii'),
    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
    ('viper', 'mark ii'), ('viper', 'mark iii')
]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20]]
```
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)

<table>
<thead>
<tr>
<th></th>
<th>max_speed</th>
<th>shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>cobra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark i</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>mark ii</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>sidewinder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark i</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>mark ii</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>viper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark ii</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>mark iii</td>
<td>16</td>
<td>36</td>
</tr>
</tbody>
</table>

Single label. Note this returns a DataFrame with a single index.

>>> df.loc['cobra']

<table>
<thead>
<tr>
<th></th>
<th>max_speed</th>
<th>shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark i</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>mark ii</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Single index tuple. Note this returns a Series.

>>> df.loc[('cobra', 'mark ii')]

max_speed 0
shield 4
Name: (cobra, mark ii), dtype: int64

Single label for row and column. Similar to passing in a tuple, this returns a Series.

>>> df.loc['cobra', 'mark ii']

max_speed 12
shield 2
Name: (cobra, mark ii), dtype: int64

Single tuple. Note using [[]] returns a DataFrame.

>>> df.loc[['cobra', 'mark ii']]

<table>
<thead>
<tr>
<th></th>
<th>max_speed</th>
<th>shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>cobra mark ii</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Single tuple for the index with a single label for the column

>>> df.loc[['cobra', 'mark ii'], 'shield']

2

Slice from index tuple to single label

>>> df.loc[['cobra', 'mark ii':'viper']]

<table>
<thead>
<tr>
<th></th>
<th>max_speed</th>
<th>shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>cobra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark i</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>mark ii</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>sidewinder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark i</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>mark ii</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>viper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark ii</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>mark iii</td>
<td>16</td>
<td>36</td>
</tr>
</tbody>
</table>
Slice from index tuple to index tuple

>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]

<table>
<thead>
<tr>
<th></th>
<th>max_speed</th>
<th>shield</th>
</tr>
</thead>
<tbody>
<tr>
<td>cobra mark i</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>mark ii</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>sidewinder mark i</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>mark ii</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>viper mark ii</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.ndim**

*property* `DataFrame.ndim`

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

*See also:*

`ndarray.ndim` Number of array dimensions.

**Examples**

```python
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

**pandas.DataFrame.shape**

*property* `DataFrame.shape`

Return a tuple representing the dimensionality of the DataFrame.

*See also:*

`ndarray.shape` Tuple of array dimensions.

**Examples**

```python
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4], 'col3': [5, 6]})
>>> df.shape
(2, 3)
```
pandas.DataFrame.size

property DataFrame.size

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also:

ndarray.size Number of elements in the array.

Examples

```python
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

pandas.DataFrame.style

property DataFrame.style

Returns a Styler object.

Contains methods for building a styled HTML representation of the DataFrame.

See also:

io.formats.style.Styler Helps style a DataFrame or Series according to the data with HTML and CSS.

pandas.DataFrame.values

property DataFrame.values

Return a Numpy representation of the DataFrame.

Warning: We recommend using DataFrame.to_numpy() instead.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

See also:

DataFrame.to_numpy Recommended alternative to this method.

DataFrame.index Retrieve the index labels.
**DataFrame.columns** Retrieving the column names.

**Notes**

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

**Examples**

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```python
>>> df = pd.DataFrame({'age': [3, 29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0   3     94      31
1  29    170     115
>>> df.dtypes
age   int64
height int64
weight int64
dtype: object
>>> df.values
array([[ 3, 94, 31],
       [29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns (e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```python
>>> df2 = pd.DataFrame([('parrot', 24.0, 'second'),
...                     ('lion', 80.5, 1),
...                     ('monkey', np.nan, None),
...                     columns=['name', 'max_speed', 'rank'])
>>> df2.dtypes
name           object
max_speed      float64
rank           object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs()</code></td>
<td>Return a Series/DataFrame with absolute numeric value of each element.</td>
</tr>
<tr>
<td><code>add(other[, axis, level, fill_value])</code></td>
<td>Get Addition of dataframe and other, element-wise (binary operator <code>add</code>).</td>
</tr>
<tr>
<td><code>add_prefix(prefix)</code></td>
<td>Prefix labels with string <code>prefix</code>.</td>
</tr>
<tr>
<td><code>add_suffix(suffix)</code></td>
<td>Suffix labels with string <code>suffix</code>.</td>
</tr>
<tr>
<td><code>agg([func, axis])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>aggregate([func, axis])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>align(other[, join, axis, level, copy, ...])</code></td>
<td>Align two objects on their axes with the specified join method.</td>
</tr>
<tr>
<td><code>all([axis, bool_only, skipna, level])</code></td>
<td>Return whether all elements are True, potentially over an axis.</td>
</tr>
<tr>
<td><code>any([axis, bool_only, skipna, level])</code></td>
<td>Return whether any element is True, potentially over an axis.</td>
</tr>
<tr>
<td><code>append(other[, ignore_index, ...])</code></td>
<td>(DEPRECATED) Append rows of <code>other</code> to the end of caller, returning a new object.</td>
</tr>
<tr>
<td><code>apply(func[, axis, raw, result_type, args])</code></td>
<td>Apply a function along an axis of the DataFrame.</td>
</tr>
<tr>
<td><code>applymap(func[, na_action])</code></td>
<td>Apply a function to a DataFrame elementwise.</td>
</tr>
<tr>
<td><code>asfreq(freq[, method, how, normalize, ...])</code></td>
<td>Convert time series to specified frequency.</td>
</tr>
<tr>
<td><code>asof(where[, subset])</code></td>
<td>Return the last row(s) without any NaNs before <code>where</code>.</td>
</tr>
<tr>
<td><code>assign(**kwargs)</code></td>
<td>Assign new columns to a DataFrame.</td>
</tr>
<tr>
<td><code>astype(dtype[, copy, errors])</code></td>
<td>Cast a pandas object to a specified dtype <code>dtype</code>.</td>
</tr>
<tr>
<td><code>at_time(time[, asof, axis])</code></td>
<td>Select values at particular time of day (e.g., 9:30AM).</td>
</tr>
<tr>
<td><code>backfill([axis, inplace, limit, downcast])</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
</tr>
<tr>
<td><code>between_time(start_time, end_time[, ...])</code></td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
</tr>
<tr>
<td><code>bfill([axis, inplace, limit, downcast])</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
</tr>
<tr>
<td><code>bool()</code></td>
<td>Return the bool of a single element Series or DataFrame.</td>
</tr>
<tr>
<td><code>boxplot([column, by, ax, fontsize, rot, ...])</code></td>
<td>Make a box plot from DataFrame columns.</td>
</tr>
<tr>
<td><code>clip([lower, upper, axis, inplace])</code></td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><code>combine(other, func[, fill_value, overwrite])</code></td>
<td>Perform column-wise combine with another DataFrame.</td>
</tr>
<tr>
<td><code>combine_first(other)</code></td>
<td>Update null elements with value in the same location in <code>other</code>.</td>
</tr>
<tr>
<td><code>compare(other[, align_axis, keep_shape, ...])</code></td>
<td>Compare to another DataFrame and show the differences.</td>
</tr>
<tr>
<td><code>convert_dtypes([infer_objects, ...])</code></td>
<td>Convert columns to best possible dtypes using dtypes supporting <code>pd.NA</code>.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Make a copy of this object's indices and data.</td>
</tr>
<tr>
<td><code>corr([method, min_periods])</code></td>
<td>Compute pairwise correlation of columns, excluding NA/null values.</td>
</tr>
<tr>
<td><code>corrwith(other[, axis, drop, method])</code></td>
<td>Compute pairwise correlation.</td>
</tr>
<tr>
<td><code>count([axis, level, numeric_only])</code></td>
<td>Count non-NA cells for each column or row.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cov</strong></td>
<td>Compute pairwise covariance of columns, excluding NA/null values.</td>
</tr>
<tr>
<td><strong>cummax</strong></td>
<td>Return cumulative maximum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>cummin</strong></td>
<td>Return cumulative minimum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>cumprod</strong></td>
<td>Return cumulative product over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>cumsum</strong></td>
<td>Return cumulative sum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>describe</strong></td>
<td>Generate descriptive statistics.</td>
</tr>
<tr>
<td><strong>diff</strong></td>
<td>First discrete difference of element.</td>
</tr>
<tr>
<td><strong>div</strong></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><strong>divide</strong></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>truediv</code>).</td>
</tr>
<tr>
<td><strong>dot</strong></td>
<td>Compute the matrix multiplication between the DataFrame and other.</td>
</tr>
<tr>
<td><strong>drop</strong></td>
<td>Drop specified labels from rows or columns.</td>
</tr>
<tr>
<td><strong>drop_duplicates</strong></td>
<td>Return DataFrame with duplicate rows removed.</td>
</tr>
<tr>
<td><strong>droplevel</strong></td>
<td>Return Series/DataFrame with requested index / column level(s) removed.</td>
</tr>
<tr>
<td><strong>dropna</strong></td>
<td>Remove missing values.</td>
</tr>
<tr>
<td><strong>duplicated</strong></td>
<td>Return boolean Series denoting duplicate rows.</td>
</tr>
<tr>
<td><strong>eq</strong></td>
<td>Get Equal to of dataframe and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
<tr>
<td><strong>equals</strong></td>
<td>Test whether two objects contain the same elements.</td>
</tr>
<tr>
<td><strong>eval</strong></td>
<td>Evaluate a string describing operations on DataFrame columns.</td>
</tr>
<tr>
<td><strong>ewm</strong></td>
<td>Provide exponentially weighted (EW) calculations.</td>
</tr>
<tr>
<td><strong>expanding</strong></td>
<td>Provide expanding window calculations.</td>
</tr>
<tr>
<td><strong>explode</strong></td>
<td>Transform each element of a list-like to a row, replicating index values.</td>
</tr>
<tr>
<td><strong>ffill</strong></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='ffill'.</td>
</tr>
<tr>
<td><strong>fillna</strong></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>Subset the dataframe rows or columns according to the specified index labels.</td>
</tr>
<tr>
<td><strong>first</strong></td>
<td>Select initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><strong>first_valid_index</strong></td>
<td>Return index for first non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td><strong>floordiv</strong></td>
<td>Get Integer division of dataframe and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><strong>from_dict</strong></td>
<td>Construct DataFrame from dict of array-like or dicts.</td>
</tr>
<tr>
<td><strong>from_records</strong></td>
<td>Convert structured or record ndarray to DataFrame.</td>
</tr>
<tr>
<td><strong>ge</strong></td>
<td>Get Greater than or equal to of dataframe and other, element-wise (binary operator <code>ge</code>).</td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>Get item from object for given key (ex: DataFrame column).</td>
</tr>
</tbody>
</table>

continues on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupby</td>
<td>Group DataFrame using a mapper or by a Series of columns.</td>
</tr>
<tr>
<td>gt</td>
<td>Get Greater than of dataframe and other, element-wise (binary operator gt).</td>
</tr>
<tr>
<td>head</td>
<td>Return the first n rows.</td>
</tr>
<tr>
<td>hist</td>
<td>Make a histogram of the DataFrame's columns.</td>
</tr>
<tr>
<td>idxmax</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td>idxmin</td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td>infer_objects</td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td>info</td>
<td>Print a concise summary of a DataFrame.</td>
</tr>
<tr>
<td>insert</td>
<td>Insert column into DataFrame at specified location.</td>
</tr>
<tr>
<td>interpolate</td>
<td>Fill NaN values using an interpolation method.</td>
</tr>
<tr>
<td>isin</td>
<td>Whether each element in the DataFrame is contained in values.</td>
</tr>
<tr>
<td>isnan</td>
<td>Detect missing values.</td>
</tr>
<tr>
<td>isnull</td>
<td>DataFrame.isnull is an alias for DataFrame.isna.</td>
</tr>
<tr>
<td>keys</td>
<td>Iterate over (column name, Series) pairs.</td>
</tr>
<tr>
<td>iteritems</td>
<td>Iterate over (column name, Series) pairs.</td>
</tr>
<tr>
<td>itertuples</td>
<td>Iterate over DataFrame rows as (index, Series) pairs.</td>
</tr>
<tr>
<td>join</td>
<td>Join columns of another DataFrame.</td>
</tr>
<tr>
<td>keys()</td>
<td>Get the 'info axis' (see Indexing for more).</td>
</tr>
<tr>
<td>kurt</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>kurtosis</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>last</td>
<td>Select final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>last_valid_index</td>
<td>Return index for last non-NA value or None, if no</td>
</tr>
<tr>
<td>le</td>
<td>Get Less than or equal to of dataframe and other, element-wise (binary operator le).</td>
</tr>
<tr>
<td>lookup</td>
<td>(DEPRECATED) Label-based &quot;fancy indexing&quot; function for DataFrame.</td>
</tr>
<tr>
<td>lt</td>
<td>Get Less than of dataframe and other, element-wise (binary operator lt).</td>
</tr>
<tr>
<td>mad</td>
<td>Return the mean absolute deviation of the values over the requested axis.</td>
</tr>
<tr>
<td>mask</td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td>max</td>
<td>Return the maximum of the values over the requested axis.</td>
</tr>
<tr>
<td>mean</td>
<td>Return the mean of the values over the requested axis.</td>
</tr>
<tr>
<td>median</td>
<td>Return the median of the values over the requested axis.</td>
</tr>
<tr>
<td>melt</td>
<td>Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.</td>
</tr>
<tr>
<td>memory_usage</td>
<td>Return the memory usage of each column in bytes.</td>
</tr>
<tr>
<td>merge</td>
<td>Merge DataFrame or named Series objects with a database-style join.</td>
</tr>
<tr>
<td>min</td>
<td>Return the minimum of the values over the requested axis.</td>
</tr>
</tbody>
</table>

continues on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mod(other[, axis, level, fill_value])</code></td>
<td>Get Modulo of dataframe and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>mode([axis, numeric_only, dropna])</code></td>
<td>Get the mode(s) of each element along the selected axis.</td>
</tr>
<tr>
<td><code>mul(other[, axis, level, fill_value])</code></td>
<td>Get Multiplication of dataframe and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>multiply(other[, axis, level, fill_value])</code></td>
<td>Get Multiplication of dataframe and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>ne(other[, axis, level])</code></td>
<td>Get Not equal to of dataframe and other, element-wise (binary operator <code>ne</code>).</td>
</tr>
<tr>
<td><code>nlargest(n, columns[, keep])</code></td>
<td>Return the first n rows ordered by columns in descending order.</td>
</tr>
<tr>
<td><code>notna()</code></td>
<td>Detect existing (non-missing) values.</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>DataFrame.notnull is an alias for DataFrame.notna.</td>
</tr>
<tr>
<td><code>nsmallest(n, columns[, keep])</code></td>
<td>Return the first n rows ordered by columns in ascending order.</td>
</tr>
<tr>
<td><code>unique([axis, dropna])</code></td>
<td>Count number of distinct elements in specified axis.</td>
</tr>
<tr>
<td><code>pad([axis, inplace, limit, downcast])</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='ffill'.</td>
</tr>
<tr>
<td><code>pct_change([periods, fill_method, limit, freq])</code></td>
<td>Percentage change between the current and a prior element.</td>
</tr>
<tr>
<td><code>pipe(func, *args, **kwargs)</code></td>
<td>Apply chainable functions that expect Series or DataFrames.</td>
</tr>
<tr>
<td><code>pivot([index, columns, values])</code></td>
<td>Return reshaped DataFrame organized by given index / column values.</td>
</tr>
<tr>
<td><code>pivot_table([values, index, columns, ...])</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>alias of pandas.plotting._core.PlotAccessor</td>
</tr>
<tr>
<td><code>pop(item)</code></td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td><code>pow(other[, axis, level, fill_value])</code></td>
<td>Get Exponential power of dataframe and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>prod([axis, skipna, level, numeric_only, ...])</code></td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td><code>product([axis, skipna, level, numeric_only, ...])</code></td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td><code>quantile([q, axis, numeric_only, interpolation])</code></td>
<td>Return values at the given quantile over requested axis.</td>
</tr>
<tr>
<td><code>query(expr[, inplace])</code></td>
<td>Query the columns of a DataFrame with a boolean expression.</td>
</tr>
<tr>
<td><code>radd(other[, axis, level, fill_value])</code></td>
<td>Get Addition of dataframe and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>rank([axis, method, numeric_only, ...])</code></td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td><code>rdiv(other[, axis, level, fill_value])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>reindex([labels, index, columns, axis, ...])</code></td>
<td>Conform Series/DataFrame to new index with optional filling logic.</td>
</tr>
<tr>
<td><code>reindex_like(other[, method, copy, limit, ...])</code></td>
<td>Return an object with matching indices as other object.</td>
</tr>
<tr>
<td><code>rename([mapper, index, columns, axis, copy, ...])</code></td>
<td>Alter axes labels.</td>
</tr>
<tr>
<td><code>rename_axis([mapper, index, columns, axis, ...])</code></td>
<td>Set the name of the axis for the index or columns.</td>
</tr>
</tbody>
</table>

continues on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reorder_levels(order[, axis])</td>
<td>Rearrange index levels using input order.</td>
</tr>
<tr>
<td>replace(to_replace, value, inplace, limit, ...)</td>
<td>Replace values given in to_replace with value.</td>
</tr>
<tr>
<td>resample(rule[, axis, closed, label, ...])</td>
<td>Resample time-series data.</td>
</tr>
<tr>
<td>reset_index([level, drop, inplace, ...])</td>
<td>Reset the index, or a level of it.</td>
</tr>
<tr>
<td>rfloordiv(other[, axis, level, fill_value])</td>
<td>Get Integer division of dataframe and other, element-wise (binary operator rfloordiv).</td>
</tr>
<tr>
<td>rmod(other[, axis, level, fill_value])</td>
<td>Get Modulo of dataframe and other, element-wise (binary operator rmod).</td>
</tr>
<tr>
<td>rmul(other[, axis, level, fill_value])</td>
<td>Get Multiplication of dataframe and other, element-wise (binary operator rmul).</td>
</tr>
<tr>
<td>rolling(window[, min_periods, center, ...])</td>
<td>Provide rolling window calculations.</td>
</tr>
<tr>
<td>round([decimals])</td>
<td>Round a DataFrame to a variable number of decimal places.</td>
</tr>
<tr>
<td>rpow(other[, axis, level, fill_value])</td>
<td>Get Exponential power of dataframe and other, element-wise (binary operator rpow).</td>
</tr>
<tr>
<td>rsub(other[, axis, level, fill_value])</td>
<td>Get Subtraction of dataframe and other, element-wise (binary operator rsub).</td>
</tr>
<tr>
<td>rtruediv(other[, axis, level, fill_value])</td>
<td>Get Floating division of dataframe and other, element-wise (binary operator rtruediv).</td>
</tr>
<tr>
<td>sample([n, frac, replace, weights, ...])</td>
<td>Return a random sample of items from an axis of object.</td>
</tr>
<tr>
<td>select_dtypes([include, exclude])</td>
<td>Return a subset of the DataFrame's columns based on the column dtypes.</td>
</tr>
<tr>
<td>sem([axis, skipna, level, ddof, numeric_only])</td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td>set_axis(labels[, axis, inplace])</td>
<td>Assign desired index to given axis.</td>
</tr>
<tr>
<td>set_flags(*[, copy, allows_duplicate_labels])</td>
<td>Return a new object with updated flags.</td>
</tr>
<tr>
<td>set_index(keys[, drop, append, inplace, ...])</td>
<td>Set the DataFrame index using existing columns.</td>
</tr>
<tr>
<td>shift([periods, freq, axis, fill_value])</td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
<tr>
<td>skew([axis, skipna, level, numeric_only])</td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td>slice_shift([periods, axis])</td>
<td>(DEPRECATED) Equivalent to shift without copying data.</td>
</tr>
<tr>
<td>sort_index([axis, level, ascending, ...])</td>
<td>Sort object by labels (along an axis).</td>
</tr>
<tr>
<td>sort_values(by[, axis, ascending, inplace, ...])</td>
<td>Sort by the values along either axis.</td>
</tr>
<tr>
<td>sparse</td>
<td>alias of pandas.core.arrays.sparse.accessor.SparseFrameAccessor</td>
</tr>
<tr>
<td>squeeze([axis])</td>
<td>Squeeze 1 dimensional axis objects into scalars.</td>
</tr>
<tr>
<td>stack([level, dropna])</td>
<td>Stack the prescribed level(s) from columns to index.</td>
</tr>
<tr>
<td>std([axis, skipna, level, ddof, numeric_only])</td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td>sub(other[, axis, level, fill_value])</td>
<td>Get Subtraction of dataframe and other, element-wise (binary operator sub).</td>
</tr>
<tr>
<td>subtract(other[, axis, level, fill_value])</td>
<td>Get Subtraction of dataframe and other, element-wise (binary operator sub).</td>
</tr>
<tr>
<td>sum([axis, skipna, level, numeric_only, ...])</td>
<td>Return the sum of the values over the requested axis.</td>
</tr>
<tr>
<td>swapaxes(axis1, axis2[, copy])</td>
<td>Interchange axes and swap values axes appropriately.</td>
</tr>
<tr>
<td>swaplevel([i, j, axis])</td>
<td>Swap levels i and j in a MultiIndex.</td>
</tr>
<tr>
<td>tail([n])</td>
<td>Return the last n rows.</td>
</tr>
<tr>
<td>take(indices[, axis, is_copy])</td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
</tbody>
</table>

continues on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_clipboard([excel, sep])</code></td>
<td>Copy object to the system clipboard.</td>
</tr>
<tr>
<td><code>to_csv([path_or_buf, sep, na_rep, ...])</code></td>
<td>Write object to a comma-separated values (csv) file.</td>
</tr>
<tr>
<td><code>to_dict([orient, into])</code></td>
<td>Convert the DataFrame to a dictionary.</td>
</tr>
<tr>
<td><code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code></td>
<td>Write object to an Excel sheet.</td>
</tr>
<tr>
<td><code>to_feather(path, **kwargs)</code></td>
<td>Write a DataFrame to the binary Feather format.</td>
</tr>
<tr>
<td><code>to_gbq(destination_table[, project_id, ...])</code></td>
<td>Write a DataFrame to a Google BigQuery table.</td>
</tr>
<tr>
<td><code>to_hdf(path_or_buf, key[, mode, complevel, ...])</code></td>
<td>Write the contained data to an HDF5 file using HDFStore.</td>
</tr>
<tr>
<td><code>to_html([buf, columns, col_space, header, ...])</code></td>
<td>Render a DataFrame as an HTML table.</td>
</tr>
<tr>
<td><code>to_json([path_or_buf, orient, date_format, ...])</code></td>
<td>Convert the object to a JSON string.</td>
</tr>
<tr>
<td><code>to_latex([buf, columns, col_space, header, ...])</code></td>
<td>Render object to a LaTeX tabular, longtable, or nested table.</td>
</tr>
<tr>
<td><code>to_markdown([buf, mode, columns, col_space, header, ...])</code></td>
<td>Print DataFrame in Markdown-friendly format.</td>
</tr>
<tr>
<td><code>to_numpy([dtype, copy, na_value])</code></td>
<td>Convert the DataFrame to a NumPy array.</td>
</tr>
<tr>
<td><code>to_period([freq, axis])</code></td>
<td>Convert DataFrame from DatetimeIndex to PeriodIndex.</td>
</tr>
<tr>
<td><code>to_p pickle([path[, compression, protocol, ...])</code></td>
<td>Pickle (serialize) object to file.</td>
</tr>
<tr>
<td><code>to_records([index, column_dtypes, index_dtypes])</code></td>
<td>Convert DataFrame to a NumPy record array.</td>
</tr>
<tr>
<td><code>to_sql([name, con[, schema, if_exists, ...])</code></td>
<td>Write records stored in a DataFrame to a SQL database.</td>
</tr>
<tr>
<td><code>to_stata([path[, convert_dates, write_index, ...])</code></td>
<td>Export DataFrame object to Stata dta format.</td>
</tr>
<tr>
<td><code>to_string([buf, columns, col_space, header, ...])</code></td>
<td>Render a DataFrame to a console-friendly tabular output.</td>
</tr>
<tr>
<td><code>to_timestamp([freq, how, axis, copy])</code></td>
<td>Cast to DatetimeIndex of timestamps, at beginning of period.</td>
</tr>
<tr>
<td><code>to_xarray()</code></td>
<td>Return an xarray object from the pandas object.</td>
</tr>
<tr>
<td><code>to_xml([path_or_buffer, index, root_name, ...])</code></td>
<td>Render a DataFrame to an XML document.</td>
</tr>
<tr>
<td><code>transform(func[, axis])</code></td>
<td>Call func on self producing a DataFrame with the same axis shape as self.</td>
</tr>
<tr>
<td><code>transpose(*args[, copy])</code></td>
<td>Transpose index and columns.</td>
</tr>
<tr>
<td><code>truediv(other[, axis, level, fill_value])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator truediv).</td>
</tr>
<tr>
<td><code>truncate([before, after, axis, copy])</code></td>
<td>Truncate a Series or DataFrame before and after some index value.</td>
</tr>
<tr>
<td><code>tsort([periods, freq, axis])</code></td>
<td>(DEPRECATED) Shift the time index, using the index’s frequency if available.</td>
</tr>
<tr>
<td><code>tz_convert(tz[, axis, level, copy])</code></td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td><code>tz_localize(tz[, axis, level, copy, ...])</code></td>
<td>Localize tz-naive index of a Series or DataFrame to target time zone.</td>
</tr>
<tr>
<td><code>unstack([level, fill_value])</code></td>
<td>Pivot a level of the (necessarily hierarchical) index labels.</td>
</tr>
<tr>
<td><code>update(other[, join, overwrite, ...])</code></td>
<td>Modify in place using non-NA values from another DataFrame.</td>
</tr>
<tr>
<td><code>value_counts([subset, normalize, sort, ...])</code></td>
<td>Return a Series containing counts of unique rows in the DataFrame.</td>
</tr>
<tr>
<td><code>var([axis, skipna, level, ddof, numeric_only])</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>where(cond[, other, inplace, axis, level, ...])</code></td>
<td>Replace values where the condition is False.</td>
</tr>
<tr>
<td><code>xs(key[, axis, level, drop_level])</code></td>
<td>Return cross-section from the Series/DataFrame.</td>
</tr>
</tbody>
</table>

3.4. DataFrame 1643
pandas.DataFrame.abs

DataFrame.abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute Calculate the absolute value element-wise.

Notes

For complex inputs, 1.2 + 1j, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```python
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```python
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```python
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0    1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```python
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
   a  b  c
0  4  10 -50
1  5  20  50
2  6  30 -30
3  7  40 100
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> df.loc[(df.c - 43).abs().argsort()]
      a    b    c
0     4    10  100
1     5    20   50
2     6    30  -30
3     7    40  -50

pandas.DataFrame.add

DataFrame.add(other, axis='columns', level=None, fill_value=None)

Get Addition of dataframe and other, element-wise (binary operator add).

Equivalent to dataframe + other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, radd.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360],
...                     index=['circle', 'triangle', 'rectangle'])
```

```
 angles  degrees
 circle    0    360
 triangle   3    180
 rectangle  4    360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
```

```
 angles  degrees
 circle    1    361
 triangle   4    181
 rectangle  5    361
```

```python
>>> df.add(1)
```

```
 angles  degrees
 circle    1    361
 triangle   4    181
 rectangle  5    361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
```

```
 angles  degrees
 circle    0.0   36.0
 triangle   0.3   18.0
 rectangle  0.4   36.0
```

```python
>>> df.rdiv(10)
```

```
 angles  degrees
 circle    inf  0.027778
 triangle  3.333333  0.055556
 rectangle 2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
```

```
 angles  degrees
 circle   -1   358
 triangle   2   178
 rectangle   3   358
```
Multiply a DataFrame of different shape with operator version.

```python
>> other = pd.DataFrame({'angles': [0, 3, 4]},
                        index=['circle', 'triangle', 'rectangle'])
>> other
angles
circle  0
triangle 3
rectangle 4

>> df * other
angles degrees
circle 0 NaN
triangle 9 NaN
rectangle 16 NaN

>> df.mul(other, fill_value=0)
angles degrees
circle 0 0.0
triangle 9 0.0
rectangle 16 0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                                'degrees': [360, 180, 360, 360, 540, 720],
                                index=['A', 'A', 'A', 'B', 'B', 'B'],
                                ['circle', 'triangle', 'rectangle',
                                 'square', 'pentagon', 'hexagon']})
>>> df_multindex
angles degrees
A circle 0 360
triangle 3 180
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720
```
>>> df.div(df_multindex, level=1, fill_value=0)

<table>
<thead>
<tr>
<th>angles</th>
<th>degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>circle</td>
</tr>
<tr>
<td></td>
<td>triangle</td>
</tr>
<tr>
<td></td>
<td>rectangle</td>
</tr>
<tr>
<td>B</td>
<td>square</td>
</tr>
<tr>
<td></td>
<td>pentagon</td>
</tr>
<tr>
<td></td>
<td>hexagon</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.add_prefix**

**DataFrame.add_prefix** *(prefix)*

Prefix labels with string `prefix`.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters**

- `prefix` [str] The string to add before each label.

**Returns**

- Series or DataFrame New Series or DataFrame with updated labels.

**See also:**

- *Series.add_suffix* Suffix row labels with string `suffix`.
- *DataFrame.add_suffix* Suffix column labels with string `suffix`.

**Examples**

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64

>>> s.add_prefix('item_')
item_0  1
item_1  2
item_2  3
item_3  4
dtype: int64

>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [3, 4, 5, 6]})
>>> df
     A  B
0  1  3
1  2  4
```
3.4. DataFrame
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> df.add_suffix('_col')
   A_col  B_col
0     1     3
1     2     4
2     3     5
3     4     6

pandas.DataFrame.agg

DataFrame.agg(func=None, axis=0, *args, **kwargs)
Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:
• function
• string function name
• list of functions and/or function names, e.g. [np.sum, 'mean']
• dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or ‘index’: apply function to each column. If 1 or ‘columns’: apply function to each row.

*args Positional arguments to pass to func.

**kwargs Keyword arguments to pass to func.

Returns

scalar, Series or DataFrame The return can be:
• scalar : when Series.agg is called with single function
• Series : when DataFrame.agg is called with a single function
• DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from

numpy aggregation functions (mean, median, prod, sum, std, var), where the default is to compute the aggregation of the flattened array, e.g., numpy.mean(arr_2d) as opposed to numpy.mean(arr_2d, axis=0).
**agg** is an alias for **aggregate**. Use the alias.

See also:

- **DataFrame.apply** Perform any type of operations.
- **DataFrame.transform** Perform transformation type operations.
- **core.groupby.GroupBy** Perform operations over groups.
- **core.resample.Resampler** Perform operations over resampled bins.
- **core.window.Rolling** Perform operations over rolling window.
- **core.window.Expanding** Perform operations over expanding window.
- **core.window.ExponentialMovingWindow** Perform operation over exponential weighted window.

**Notes**

`agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```python
>>> df = pd.DataFrame([[1, 2, 3],
...                     [4, 5, 6],
...                     [7, 8, 9],
...                     [np.nan, np.nan, np.nan]],
...                     columns=['A', 'B', 'C'])

Aggregate these functions over the rows.

```python
>>> df.agg(['sum', 'min'])
   A   B   C
sum 12.0 15.0 18.0
min  1.0  2.0  3.0
```

Different aggregations per column.

```python
>>> df.agg({'A': ['sum', 'min'], 'B': ['min', 'max']})
   A   B
sum 12.0 NaN
min  1.0  2.0
max NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```python
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
   A   B   C
x  7.0 NaN NaN
y  NaN  2.0 NaN
z  NaN  NaN  6.0
```
Aggregate over the columns.

```python
df.agg("mean", axis="columns")
```

0  2.0  
1  5.0  
2  8.0  
3  NaN  
dtype: float64

**pandas.DataFrame.aggregate**

DataFrame.aggregate(func=None, axis=0, *args, **kwargs)

Aggregate using one or more operations over the specified axis.

**Parameters**

- `func` [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.
  
  Accepted combinations are:
  
  • function
  • string function name
  • list of functions and/or function names, e.g. `[np.sum, 'mean']`
  • dict of axis labels -> functions, function names or list of such.

- `axis` [{0 or 'index', 1 or 'columns'}, default 0] If 0 or ‘index’: apply function to each column. If 1 or ‘columns’: apply function to each row.

  * `args` Positional arguments to pass to `func`.

  * `**kwargs` Keyword arguments to pass to `func`.

**Returns**

- scalar, Series or DataFrame  The return can be:
  
  • scalar : when Series agg is called with single function
  • Series : when DataFrame agg is called with a single function
  • DataFrame : when DataFrame agg is called with several functions

Return scalar, Series or DataFrame.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from `numpy` aggregation functions (`mean`, `median`, `prod`, `sum`, `std`, `var`), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0).

`agg` is an alias for `aggregate`. Use the alias.

See also:
**DataFrame.apply** Perform any type of operations.

**DataFrame.transform** Perform transformation type operations.

- `core.groupby.GroupBy` Perform operations over groups.
- `core.resample.Resampler` Perform operations over resampled bins.
- `core.window.Rolling` Perform operations over rolling window.
- `core.window.Expanding` Perform operations over expanding window.
- `core.window.ExponentialMovingWindow` Perform operation over exponential weighted window.

**Notes**

`agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

A passed user-defined function will be passed a Series for evaluation.

**Examples**

```python
>>> df = pd.DataFrame([[1, 2, 3],
...                     [4, 5, 6],
...                     [7, 8, 9],
...                     [np.nan, np.nan, np.nan]],
...                     columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```python
>>> df.agg(['sum', 'min'])
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>12.0</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td>min</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Different aggregations per column.

```python
>>> df.agg({'A': ['sum', 'min'], 'B': ['min', 'max']})
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>12.0</td>
<td>NaN</td>
</tr>
<tr>
<td>min</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>max</td>
<td>NaN</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```python
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7.0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>y</td>
<td>NaN</td>
<td>2.0</td>
<td>NaN</td>
</tr>
<tr>
<td>z</td>
<td>NaN</td>
<td>NaN</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Aggregate over the columns.
```python
>>> df.agg("mean", axis="columns")
0   2.0
1   5.0
2   8.0
3   NaN
dtype: float64
```

### pandas.DataFrame.align

DataFrame.align(\texttt{other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0, broadcast_axis=None})

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

**Parameters**

- **other** [DataFrame or Series]
- **join** [{‘outer’, ‘inner’, ‘left’, ‘right’}, default ‘outer’]
- **axis** [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).
- **level** [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **copy** [bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.
- **fill_value** [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any “compatible” value.
- **method** [{‘backfill’, ‘bfill’, ‘pad’, ‘ffill’}, None], default None Method to use for filling holes in reindexed Series:
  - pad / ffill: propagate last valid observation forward to next valid.
  - backfill / bfill: use NEXT valid observation to fill gap.
- **limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.
- **fill_axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Filling axis, method and limit.
- **broadcast_axis** [{0 or ‘index’, 1 or ‘columns’}, default None] Broadcast values along this axis, if aligning two objects of different dimensions.

**Returns**

- \((\text{left, right})\) [\{(DataFrame, type of other)\}] Aligned objects.
Examples

```python
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=['D', 'B', 'E', 'A'], index=[1, 2])
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=['A', 'B', 'C', 'D'],
...     index=[2, 3, 4])
... )
>>> df
A  B  C  D
1 1  2  3  4
2 6  7  8  9

Align on columns:

```python
>>> left, right = df.align(other, join='outer', axis=1)
>>> left
A  B  C  D  E
1 4  2 NaN  1  3
2 9  7 NaN  6  8

```python
right
A  B  C  D
2 10  20 30  40 NaN
3 60  70  80  90 NaN
4 600 700 800 900 NaN

We can also align on the index:

```python
>>> left, right = df.align(other, join='outer', axis=0)
>>> left
D  B  E  A
1 1.0 2.0 3.0 4.0
2 6.0 7.0 8.0 9.0
3 NaN NaN NaN NaN
4 NaN NaN NaN NaN

```
```python
right
A  B  C  D
1 NaN NaN NaN NaN
2 10.0 20.0 30.0 40.0
3 60.0 70.0 80.0 90.0
4 600.0 700.0 800.0 900.0

Finally, the default `axis=None` will align on both index and columns:

```python
>>> left, right = df.align(other, join='outer', axis=None)
>>> left
(continues on next page)
```
pandas.DataFrame.all

**DataFrame.all**(*axis=0, bool_only=None, skipna=True, level=None, **kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

**Parameters**

- **axis** [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.
  - 0 / ‘index’: reduce the index, return a Series whose index is the original column labels.
  - 1 / ‘columns’: reduce the columns, return a Series whose index is the original index.
  - None: reduce all axes, return a scalar.
- **bool_only** [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.
- **skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Series or DataFrame** If level is specified, then DataFrame is returned; otherwise, Series is returned.

**See also:**

- **Series.all** Return True if all elements are True.
- **DataFrame.any** Return True if one (or more) elements are True.
Examples

Series

```python
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```python
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1  col2
0   True   True
1   True   False
```

Default behaviour checks if column-wise values all return True.

```python
>>> df.all()
   col1  col2
0   True   True
dtype: bool
```

Specify axis='columns' to check if row-wise values all return True.

```python
>>> df.all(axis='columns')
0   True
1   False
dtype: bool
```

Or axis=None for whether every value is True.

```python
>>> df.all(axis=None)
False
```

pandas.DataFrame.any

DataFrame.any(axis=0, bool_only=None, skipna=True, level=None, **kwargs)

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters
**axis** [[0 or ‘index’, 1 or ‘columns’, None], default 0] Indicate which axis or axes should be reduced.

- 0 / ‘index’: reduce the index, return a Series whose index is the original column labels.
- 1 / ‘columns’: reduce the columns, return a Series whose index is the original index.
- None: reduce all axes, return a scalar.

**bool_only** [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

**level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Series or DataFrame** If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

- **numpy.any** Numpy version of this method.
- **Series.any** Return whether any element is True.
- **Series.all** Return whether all elements are True.
- **DataFrame.any** Return whether any element is True over requested axis.
- **DataFrame.all** Return whether all elements are True over requested axis.

**Examples**

**Series**

For Series input, the output is a scalar indicating whether any element is True.

```python
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

**DataFrame**

Whether each column contains at least one True element (the default).
```python
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0, 2], 'C': [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0

>>> df.any()
A   True
B   True
C  False
dtype: bool

Aggregating over the columns.
```

```python
>>> df = pd.DataFrame({'A': [True, False], 'B': [1, 2]})
>>> df
   A  B
0  True  1
1  False  2

>>> df.any(axis='columns')
0   True
1   True
dtype: bool
```

```python
>>> df = pd.DataFrame({'A': [True, False], 'B': [1, 0]})
>>> df
   A  B
0  True  1
1  False  0

>>> df.any(axis='columns')
0   True
1  False
dtype: bool
```

Aggregating over the entire DataFrame with axis=None.

```python
>>> df = pd.DataFrame({'A': [True, False], 'B': [1, 0]})
>>> df
   A  B
0  True  1
1  False  0

>>> df.any(axis=None)
True
```

*any* for an empty DataFrame is an empty Series.

```python
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```
**pandas.DataFrame.append**

`DataFrame.append(other, ignore_index=False, verify_integrity=False, sort=False)`

Append rows of `other` to the end of caller, returning a new object.

Deprecated since version 1.4.0: Use `concat()` instead. For further details see `Deprecated DataFrame.append and Series.append`

Columns in `other` that are not in the caller are added as new columns.

**Parameters**

- `other` [DataFrame or Series/dict-like object, or list of these] The data to append.
- `ignore_index` [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.
- `verify_integrity` [bool, default False] If True, raise `ValueError` on creating index with duplicates.
- `sort` [bool, default False] Sort columns if the columns of `self` and `other` are not aligned.

Changed in version 1.0.0: Changed to not sort by default.

**Returns**

`DataFrame` A new DataFrame consisting of the rows of caller and the rows of `other`.

**See also:**

`concat` General function to concatenate DataFrame or Series objects.

**Notes**

If a list of dict/series is passed and the keys are all contained in the DataFrame’s index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

**Examples**

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'), index=['x', 'y'])
>>> df
    A  B
   x 1  2
   y 3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'), index=['x', 'y'])
>>> df.append(df2)
    A  B
   x 1  2
   y 3  4
   x 5  6
   y 7  8
```

With `ignore_index` set to True:
The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```python
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
A
0 0
1 1
2 2
3 3
4 4
```

More efficient:

```python
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...     ignore_index=True)
A
0 0
1 1
2 2
3 3
4 4
```

**pandas.DataFrame.apply**

DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), **kwargs)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame’s index (axis=0) or the DataFrame’s columns (axis=1). By default (result_type=None), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the result_type argument.

**Parameters**

- **func** [function] Function to apply to each column or row.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Axis along which the function is applied:
  - 0 or ‘index’: apply function to each column.
  - 1 or ‘columns’: apply function to each row.
- **raw** [bool, default False] Determines if row or column is passed as a Series or ndarray object:
  - False: passes each row or column as a Series to the function.
• **True** : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

**result_type** [{‘expand’, ‘reduce’, ‘broadcast’, None}, default None] These only act when axis=1 (columns):

• ‘expand’ : list-like results will be turned into columns.
• ‘reduce’ : returns a Series if possible rather than expanding list-like results. This is the opposite of ‘expand’.
• ‘broadcast’ : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

**args** [tuple] Positional arguments to pass to *func* in addition to the array/series.

**kwargs** Additional keyword arguments to pass as keywords arguments to *func*.

**Returns**

Series or DataFrame  Result of applying *func* along the given axis of the DataFrame.

**See also:**

*DataFrame.applymap* For elementwise operations.

*DataFrame.aggregate* Only perform aggregating type operations.

*DataFrame.transform* Only perform transforming type operations.

**Notes**

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

**Examples**

```python
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```python
>>> df.apply(np.sqrt)
    A  B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis
```python
>>> df.apply(np.sum, axis=0)
A 12
B 27
dtype: int64

>>> df.apply(np.sum, axis=1)
0 13
1 13
2 13
dtype: int64

Returning a list-like will result in a Series

```python
>>> df.apply(lambda x: [1, 2], axis=1)
0 [1, 2]
1 [1, 2]
2 [1, 2]
dtype: object

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```python
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
      0  1
0    1  2
1    1  2
2    1  2

Returning a Series inside the function is similar to passing result_type='expand'. The resulting column names will be the Series index.

```python
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2

Passing result_type='broadcast' will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```python
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```
**pandas.DataFrame.applymap**

**DataFrame.applymap(func, na_action=None, **kwargs)**

Apply a function to a DataFrame elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

**Parameters**

- **func** [callable] Python function, returns a single value from a single value.
- **na_action** [{None, ‘ignore’}, default None] If ‘ignore’, propagate NaN values, without passing them to func.
  
  New in version 1.2.
- **kwargs** Additional keyword arguments to pass as keywords arguments to func.
  
  New in version 1.3.0.

**Returns**

- **DataFrame** Transformed DataFrame.

**See also:**

- **DataFrame.apply** Apply a function along input axis of DataFrame.

**Examples**

```python
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
 0 1
0 1.000 2.120
1 3.356 4.567

>>> df.applymap(lambda x: len(str(x)))
 0 1
0 3 4
1 5 5

Like Series.map, NA values can be ignored:

```python
>>> df_copy = df.copy()
>>> df_copy.iloc[0, 0] = pd.NA
>>> df_copy.applymap(lambda x: len(str(x)), na_action='ignore')
 0 1
0 <NA> 4
1 5 5
```  

Note that a vectorized version of func often exists, which will be much faster. You could square each number elementwise.

```python
>>> df.applymap(lambda x: x**2)
 0 1
0 1.000000 4.494400
1 11.262736 20.857489
```
But it’s better to avoid applymap in that case.

```python
>>> df ** 2
   0   1
0 1.000000 4.494400
1 11.262736 20.857489
```

**pandas.DataFrame.asfreq**

`DataFrame.asfreq(freq=None, method=None, how='end', normalize=False, fill_value=None)`

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this DataFrame is a `PeriodIndex`, the new index is the result of transforming the original index with `PeriodIndex.asfreq` (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to `pd.date_range(start, end, freq=freq)` where `start` and `end` are, respectively, the first and last entries in the original index (see `pandas.date_range()`). The values corresponding to any timesteps in the new index which were not present in the original index will be null (`NaN`), unless a method for filling such unknowns is provided (see the `method` parameter below).

The `resample()` method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

**Parameters**

- `freq` [DateOffset or str] Frequency DateOffset or string.
- `method` [{‘pad’ / ‘ffill’, ‘backfill’ / ‘bfill’}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):
  - ‘pad’ / ‘ffill’: propagate last valid observation forward to next valid
  - ‘backfill’ / ‘bfill’: use NEXT valid observation to fill.
- `how` [{‘start’, ‘end’}, default end] For PeriodIndex only (see PeriodIndex.asfreq).
- `normalize` [bool, default False] Whether to reset output index to midnight.
- `fill_value` [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

**Returns**

- `DataFrame` DataFrame object reindexed to the specified frequency.

**See also:**

- `reindex` Conform DataFrame to new index with optional filling logic.
Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
   s
2000-01-01 00:00:00  0.0
2000-01-01 00:01:00  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:03:00  3.0
```

Upsample the series into 30 second bins.

```python
>>> df.asfreq(freq='30S')
   s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  NaN
2000-01-01 00:03:00  3.0
```

Upsample again, providing a fill value.

```python
>>> df.asfreq(freq='30S', fill_value=9.0)
   s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  9.0
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  9.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  9.0
2000-01-01 00:03:00  3.0
```

Upsample again, providing a method.

```python
>>> df.asfreq(freq='30S', method='bfill')
   s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```
pandas.DataFrame.asof

DataFrame.asof(where, subset=None)

Return the last row(s) without any NaNs before where.

The last row (for each element in where, if list) without any NaN is taken. In case of a DataFrame, the last row without NaN considering only the subset of columns (if not None)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where [date or array-like of dates] Date(s) before which the last row(s) are returned.

subset [str or array-like of str, default None] For DataFrame, if not None, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame The return can be:

• scalar : when self is a Series and where is a scalar
• Series: when self is a Series and where is an array-like, or when self is a DataFrame and where is a scalar
• DataFrame: when self is a DataFrame and where is an array-like

Return scalar, Series, or DataFrame.

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar where.

```python
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64

>>> s.asof(20)
2.0
```
For a sequence where, a Series is returned. The first value is NaN, because the first element of where is before the first index value.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> s.asof([5, 20])
5    NaN
20  2.0
dtype: float64
```

Missing values are not considered. The following is 2.0, not NaN, even though NaN is at the index location for 30.

```python
>>> s.asof(30)
2.0
```

Take all columns into consideration

```python
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
                    'b': [None, None, None, None, 500],
                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
                                            '2018-02-27 09:02:00',
                                            '2018-02-27 09:03:00',
                                            '2018-02-27 09:04:00',
                                            '2018-02-27 09:05:00'])
```  

```python
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
                             '2018-02-27 09:04:30']))
a  b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```python
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
                             '2018-02-27 09:04:30']),
          subset=['a'])
a  b
2018-02-27 09:03:30 30.0 NaN
2018-02-27 09:04:30 40.0 NaN
```

**pandas.DataFrame.assign**

DataFrame.assign(**kwargs)

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are reassigned will be overwritten.

**Parameters**

**kwargs [dict of {str: callable or Series}]** The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn’t check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

**Returns**

DataFrame A new DataFrame with the new columns in addition to all the existing columns.
Notes

Assigning multiple columns within the same assign is possible. Later items in ‘**kwargs’ may refer to newly created or modified columns in ‘df’; items are computed and assigned into ‘df’ in order.

Examples

```python
def = pd.DataFrame({'temp_c': [17.0, 25.0],
                   'index':['Portland', 'Berkeley'])
def
```

```
temp_c
Portland 17.0
Berkeley 25.0
```

Where the value is a callable, evaluated on df:

```python
def.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
```

```
temp_c temp_f
Portland 17.0 62.6
Berkeley 25.0 77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```python
def.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
```

```
temp_c temp_f
Portland 17.0 62.6
Berkeley 25.0 77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```python
def.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
           temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
```

```
temp_c temp_f temp_k
Portland 17.0 62.6 290.15
Berkeley 25.0 77.0 298.15
```

**pandas.DataFrame.astype**

DataFrames can be cast to a specified dtype using the `astype` method.

```python
def.astype(dtype, copy=True, errors='raise')
```

Cast a pandas object to a specified dtype `dtype`.

**Parameters**

- `dtype` [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast the entire pandas object to the same type. Alternatively, use `{col: dtype, ...}`, where `col` is a column label and `dtype` is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

- `copy` [bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

- `errors` ["'raise', 'ignore'", default 'raise'] Control raising of exceptions on invalid data for provided dtype.
• raise: allow exceptions to be raised
• ignore: suppress exceptions. On error return original object.

Returns

casted [same type as caller]

See also:

to_datetime Convert argument to datetime.
to_timedelta Convert argument to timedelta.
to_numeric Convert argument to a numeric type.
numpy.ndarray.astype Cast a numpy array to a specified type.

Notes

Deprecated since version 1.3.0: Using astype to convert from timezone-naive dtype to timezone-aware dtype is deprecated and will raise in a future version. Use Series.dt.tz_localize() instead.

Examples

Create a DataFrame:

```python
>>> d = { 'col1': [1, 2], 'col2': [3, 4] }
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1 int64
col2 int64
dtype: object
```

Cast all columns to int32:

```python
>>> df.astype('int32').dtypes
col1 int32
col2 int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```python
>>> df.astype({'col1': 'int32'}).dtypes
col1 int32
col2 int64
dtype: object
```

Create a series:

```python
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0   1
1   2
dtype: int32
>>> ser.astype('int64')
```

(continues on next page)
Convert to categorical type:

```
>>> ser.astype('category')
0  1
1  2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype

>>> cat_dtype = CategoricalDtype(...
   categories=[2, 1], ordered=True)

>>> ser.astype(cat_dtype)
0  1
1  2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1, 2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1
# note that s1[0] has changed too
0   10
1    2
dtype: int64
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))

>>> ser_date
0  2020-01-01
1  2020-01-02
2  2020-01-03
dtype: datetime64[ns]
```

**pandas.DataFrame.at_time**

`DataFrame.at_time(time, asof=False, axis=None)`

Select values at particular time of day (e.g., 9:30AM).

**Parameters**

- **time** [datetime.time or str]
- **axis** [[0 or ‘index’, 1 or ‘columns’], default 0]

**Returns**
Series or DataFrame

Raises

TypeError If the index is not a DatetimeIndex

See also:

between_time Select values between particular times of the day.
first Select initial periods of time series based on a date offset.
last Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time Get just the index locations for values at particular time of the day.

Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                         A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4

>>> ts.at_time('12:00')
                         A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

dataframe.backfill

DataFrame.backfill(axis=None, inplace=False, limit=None, downcast=None)

Synonym for DataFrame.fillna() with method='bfill'.

Returns

Series/DataFrame or None Object with missing values filled or None if inplace=True.

Dataframe.between_time

DataFrame.between_time(start_time, end_time, include_start=NoDefault.no_default,
                        include_end=NoDefault.no_default, inclusive=None, axis=None)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting start_time to be later than end_time, you can get the times that are not between the two times.

Parameters

start_time [datetime.time or str] Initial time as a time filter limit.
end_time [datetime.time or str] End time as a time filter limit.
**include_start** [bool, default True] Whether the start time needs to be included in the result.

Deprecated since version 1.4.0: Arguments *include_start* and *include_end* have been deprecated to standardize boundary inputs. Use *inclusive* instead, to set each bound as closed or open.

**include_end** [bool, default True] Whether the end time needs to be included in the result.

Deprecated since version 1.4.0: Arguments *include_start* and *include_end* have been deprecated to standardize boundary inputs. Use *inclusive* instead, to set each bound as closed or open.

**inclusive** [{“both”, “neither”, “left”, “right”}, default “both”] Include boundaries; whether to set each bound as closed or open.

**axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Determine range time on index or columns value.

**Returns**

*Series or DataFrame*  Data from the original object filtered to the specified dates range.

**Raises**

**TypeError**  If the index is not a *DatetimeIndex*

**See also:**

*at_time*  Select values at a particular time of the day.

*first*  Select initial periods of time series based on a date offset.

*last*  Select final periods of time series based on a date offset.

*DatetimeIndex.indexer_between_time*  Get just the index locations for values between particular times of the day.

**Examples**

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
   A
0  2018-04-09 00:00:00  1
1  2018-04-10 00:20:00  2
2  2018-04-11 00:40:00  3
3  2018-04-12 01:00:00  4

>>> ts.between_time('0:15', '0:45')
   A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting *start_time* later than *end_time*:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> ts.between_time('0:45', '0:15')  
   A
2018-04-09 00:00:00 1
2018-04-12 01:00:00 4
```

**pandas.DataFrame.bfill**

`DataFrame.bfill(axis=None, inplace=False, limit=None, downcast=None)`

Synonym for `DataFrame.fillna()` with method='bfill'.

**Returns**

- **Series/DataFrame or None** Object with missing values filled or None if inplace=True.

**pandas.DataFrame.bool**

`DataFrame.bool()`

Return the bool of a single element Series or DataFrame.

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

**Returns**

- **bool** The value in the Series or DataFrame.

**See also:**

- **Series.astype** Change the data type of a Series, including to boolean.
- **DataFrame.astype** Change the data type of a DataFrame, including to boolean.
- **numpy.bool_** NumPy boolean data type, used by pandas for boolean values.

**Examples**

The method will only work for single element objects with a boolean value:

```python
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```

```python
>>> pd.DataFrame({'col': [True]}).bool()
True
>>> pd.DataFrame({'col': [False]}).bool()
False
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

DataFrame.boxplot

DataFrame.boxplot(column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, figsize=None,
layout=None, return_type=None, backend=None, **kwargs)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. By default, they extend no more than 1.5 * IQR (IQR = Q3 - Q1) from the edges of the box, ending at the farthest data point within that interval. Outliers are plotted as separate dots.

For further details see Wikipedia’s entry for boxplot.

Parameters

- **column** [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to pandas.DataFrame.groupby().
- **by** [str or array-like, optional] Column in the DataFrame to pandas.DataFrame.groupby(). One box-plot will be done per value of columns in by.
- **ax** [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.
- **fontsize** [float or str] Tick label font size in points or as a string (e.g., large).
- **rot** [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.
- **grid** [bool, default True] Setting this to True will show the grid.
- **figsize** [A tuple (width, height) in inches] The size of the figure to create in matplotlib.
- **layout** [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.
- **return_type** [[‘axes’, ‘dict’, ‘both’] or None, default ‘axes’] The kind of object to return. The default is axes.
  - ‘axes’ returns the matplotlib axes the boxplot is drawn on.
  - ‘dict’ returns a dictionary whose values are the matplotlib Lines of the boxplot.
  - ‘both’ returns a namedtuple with the axes and dict.
  - when grouping with by, a Series mapping columns to return_type is returned.
    - If return_type is None, a NumPy array of axes with the same shape as layout is returned.
- **backend** [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend.
  - New in version 1.0.0.
- **kwargs** All other plotting keyword arguments to be passed to matplotlib.pyplot.boxplot().

Returns
result  See Notes.

See also:

Series.plot.hist  Make a histogram.
matplotlib.pyplot.boxplot  Matplotlib equivalent plot.

Notes

The return type depends on the return_type parameter:

- 'axes': object of class matplotlib.axes.Axes
- 'dict': dict of matplotlib.lines.Line2D objects
- 'both': a named tuple with structure (ax, lines)

For data grouped with by, return a Series of the above or a numpy array:

- Series
- array (for return_type = None)

Use return_type='dict' when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by df.boxplot() or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10, 4),
...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

Boxplots of variables distributions grouped by the values of a third variable can be created using the option by. For instance:

```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                   columns=['Col1', 'Col2'])
>>> boxplot = df.boxplot(by='X')
```

A list of strings (i.e. ['X', 'Y']) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

```
>>> df = pd.DataFrame(np.random.randn(10, 3),
...                   columns=['Col1', 'Col2', 'Col3'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

The layout of boxplot can be adjusted giving a tuple to layout:
3.4. DataFrame
Boxplot grouped by ['X', 'Y']

Col1

Col2

(A, A)  (A, B)  (B, A)  (B, B)
[X, Y]

(A, A)  (A, B)  (B, A)  (B, B)
[X, Y]
Boxplot grouped by X

Col1

Col2

A

B

x

Additional formatting can be done to the boxplot, like suppressing the grid (grid=False), rotating the labels in the x-axis (i.e. rot=45) or changing the fontsize (i.e. fontsize=15):

```python
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

The parameter `return_type` can be used to select the type of element returned by `boxplot`. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
```

```python
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with by, a Series mapping columns to `return_type` is returned:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
                       return_type='axes')
```

```python
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is `None`, a NumPy array of axes with the same shape as `layout` is returned:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
    ...                      return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

pandas.DataFrame.clip

DataFrame.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

- **lower** [float or array-like, default None] Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.
- **upper** [float or array-like, default None] Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.
- **axis** [int or str axis name, optional] Align object with lower and upper along the given axis.
- **inplace** [bool, default False] Whether to perform the operation in place on the data.
- **args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame or None Same type as calling object with the values outside the clip boundaries replaced or None if inplace=True.

See also:

Series.clip Trim values at input threshold in series.

DataFrame.clip Trim values at input threshold in dataframe.

numpy.clip Clip (limit) the values in an array.

Examples

```python
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0     9    -2
1    -3    -7
2     0     6
3    -1     8
4     5    -5
```

Clips per column using lower and upper thresholds:
Clips using specific lower and upper thresholds per column element:

```python
>>> df.clip(-4, 6)
col_0  col_1
0    6   -2
1   -3   -4
2     0    6
3   -1    6
4    5   -4
```

Clips using specific lower threshold per column element, with missing values:

```python
>>> t = pd.Series([2, -4, np.NaN, 6, 3])
```

```python
>>> df.clip(t, axis=0)
col_0  col_1
0    9    2
1   -3   -4
2     0    6
3    6    8
4    5    3
```
pandas.DataFrame.combine

DataFrame.combine(other, func, fill_value=None, overwrite=True)

Perform column-wise combine with another DataFrame.

Combines a DataFrame with other DataFrame using func to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other [DataFrame] The DataFrame to merge column-wise.

func [function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value [scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite [bool, default True] If True, columns in self that do not exist in other will be overwritten with NaNs.

Returns

DataFrame Combination of the provided DataFrames.

See also:

DataFrame.combine_first Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```python
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```python
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using fill_value fills Nones prior to passing the column to the merge function.

```python
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A  B
0  0  3
1  0  3
```

(continues on next page)
However, if the same element in both dataframes is None, that None is preserved

```python
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A  B
0  0 -5.0
1  0  4.0
```

Example that demonstrates the use of `overwrite` and behavior when the axis differ between the dataframes.

```python
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1]}, index=[1, 2])
>>> df1.combine(df2, take_smaller)
   A  B  C
0 NaN NaN NaN
1 NaN 3.0 -10.0
2 NaN 3.0  1.0
```

```python
>>> df1.combine(df2, take_smaller, overwrite=False)
   A  B  C
0 0.0 NaN NaN
1 0.0 3.0 -10.0
2 NaN 3.0  1.0
```

Demonstrating the preference of the passed in dataframe.

```python
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A  B  C
0 NaN NaN NaN
1 0.0 NaN NaN
2 NaN 3.0 NaN
```

```python
>>> df2.combine(df1, take_smaller, overwrite=False)
   A  B  C
0 0.0 NaN NaN
1 0.0 3.0  1.0
2 NaN 3.0  1.0
```
pandas.DataFrame.combine_first

Dataframe.combine_first(other)
Update null elements with value in the same location in other.

Combine two Dataframe objects by filling null values in one Dataframe with non-null values from other Dataframe. The row and column indexes of the resulting Dataframe will be the union of the two.

Parameters
other [Dataframe] Provided Dataframe to use to fill null values.

Returns
Dataframe The result of combining the provided Dataframe with the other object.

See also:
Dataframe.combine Perform series-wise operation on two Dataframes using a given function.

Examples

```python
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
   A  B
0  1.0 3.0
1  0.0 4.0
```

Null values still persist if the location of that null value does not exist in other

```python
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], index=[1, 2]})
>>> df1.combine_first(df2)
   A  B  C
0  NaN 4.0  NaN
1  0.0 3.0  1.0
2  NaN 3.0  1.0
```

pandas.DataFrame.compare

Dataframe.compare(other, align_axis=1, keep_shape=False, keep_equal=False)
Compare to another Dataframe and show the differences.

New in version 1.1.0.

Parameters
other [Dataframe] Object to compare with.

align_axis [{0 or ‘index’, 1 or ‘columns’}, default 1] Determine which axis to align the comparison on.

- 0, or ‘index’ [Resulting differences are stacked vertically] with rows drawn alternately from self and other.
1. or `columns`  [Resulting differences are aligned horizontally] with columns drawn alternately from `self` and `other`.

**keep_shape**  [bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

**keep_equal**  [bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

**Returns**

DataFrame  DataFrame that shows the differences stacked side by side.

The resulting index will be a MultiIndex with `self` and `other` stacked alternately at the inner level.

**Raises**

ValueError  When the two DataFrames don’t have identical labels or shape.

**See also:**

- `Series.compare`  Compare with another Series and show differences.
- `DataFrame.equals`  Test whether two objects contain the same elements.

**Notes**

Matching NaNs will not appear as a difference.

Can only compare identically-labeled (i.e. same shape, identical row and column labels) DataFrames

**Examples**

```python
>>> df = pd.DataFrame(
...     {
...         "col1": ["a", "a", "b", "b", "a"],
...         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
...         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
...     },
...     columns=["col1", "col2", "col3"],
... )

>>> df
    col1  col2  col3
0  a     1.0  1.0
1  a     2.0  2.0
2  b     3.0  3.0
3  b  NaN  4.0
4  a     5.0  5.0

>>> df2 = df.copy()
>>> df2.loc[0, 'col1'] = 'c'
>>> df2.loc[2, 'col3'] = 4.0
>>> df2
    col1  col2  col3
0  c     1.0  1.0
```

(continues on next page)
Align the differences on columns

```python
>>> df.compare(df2)
col1  col3
     self    other    self    other
0   a    c  NaN     NaN
2  NaN  NaN   3.0     4.0
```

Stack the differences on rows

```python
>>> df.compare(df2, align_axis=0)
col1  col3
     self    other    self    other
0   a    c  NaN     NaN
2  NaN  NaN   3.0     4.0
```

Keep the equal values

```python
>>> df.compare(df2, keep_equal=True)
col1  col3
     self    other    self    other
0   a    c  1.0     1.0
2   b    b  3.0     4.0
```

Keep all original rows and columns

```python
>>> df.compare(df2, keep_shape=True)
col1  col2  col3
     self    other    self    other    self    other
0   a    c  NaN     NaN  NaN     NaN
1  NaN  NaN  NaN     NaN  NaN     NaN
2  NaN  NaN  NaN     NaN  3.0     4.0
3  NaN  NaN  NaN     NaN  NaN     NaN
4  NaN  NaN  NaN     NaN  NaN     NaN
```

Keep all original rows and columns and also all original values

```python
>>> df.compare(df2, keep_shape=True, keep_equal=True)
col1  col2  col3
     self    other    self    other    self    other
0   a    c  1.0     1.0   1.0     1.0
1   a    a  2.0     2.0   2.0     2.0
2   b    b  3.0     3.0   3.0     4.0
3   b    b  NaN     NaN   4.0     4.0
4   a    a  5.0     5.0   5.0     5.0
pandas.DataFrame.convert_dtypes

DataFrame.convert_dtypes(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True)

Convert columns to best possible dtypes using dtypes supporting pd.NA.

New in version 1.0.0.

Parameters

infer_objects [bool, default True] Whether object dtypes should be converted to the best possible types.

convert_string [bool, default True] Whether object dtypes should be converted to StringDtype().

convert_integer [bool, default True] Whether, if possible, conversion can be done to integer extension types.

convert_boolean [bool, default True] Whether object dtypes should be converted to BooleanDtype().

convert_floating [bool, default True] Whether, if possible, conversion can be done to floating extension types. If convert_integer is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

New in version 1.2.0.

Returns

Series or DataFrame Copy of input object with new dtype.

See also:

infer_objects Infer dtypes of objects.
to_datetime Convert argument to datetime.
to_timedelta Convert argument to timedelta.
to_numeric Convert argument to a numeric type.

Notes

By default, convert_dtypes will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support pd.NA. By using the options convert_string, convert_integer, convert_boolean and convert_boolean, it is possible to turn off individual conversions to StringDtype, the integer extension types, BooleanDtype or floating extension types, respectively.

For object-dtyped columns, if infer_objects is True, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to StringDtype, BooleanDtype or an appropriate integer or floating extension type, otherwise leave as object.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.
In the future, as new dtypes are added that support `pd.NA`, the results of this method will change to support those new dtypes.

### Examples

```python
>>> df = pd.DataFrame(
...     {
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )

Start with a DataFrame with default dtypes.

```python
>>> df
    a   b    c    d    e    f
0  1    x True  h  10.0  NaN
1  2  y False  i   NaN  100.5
2  3  z   NaN  NaN  20.0  200.0
```  

```python
>>> df.dtypes
a    int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```python
>>> dfn = df.convert_dtypes()

```  

```python
>>> dfn
    a   b    c    d    e    f
0  1    x True  h  10   <NA>
1  2  y False  i  <NA>  100.5
2  3  z   NaN  NaN <NA>  200.0
```

```python
>>> dfn.dtypes
a    Int32
b    string
c  boolean
d  string
e    Int64
f    Float64
dtype: object
```

Start with a Series of strings and missing data represented by `np.nan`.

```python
```
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0    a
1    b
2   NaN
dtype: object

Obtain a Series with dtype StringDtype.

>>> s.convert_dtypes()
0    a
1    b
2  <NA>
dtype: string

pandas.DataFrame.copy

DataFrame.copy(deep=True)

Make a copy of this object’s indices and data.

When deep=True (default), a new object will be created with a copy of the calling object’s data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When deep=False, a new object will be created without copying the calling object’s data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

depth [bool, default True] Make a deep copy, including a copy of the data and the indices.
With deep=False neither the indices nor the data are copied.

Returns

copy [Series or DataFrame] Object type matches caller.

Notes

When deep=True, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to copy.deepcopy in the Standard Library, which recursively copies object data (see examples below).

While Index objects are copied when deep=True, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.
Examples

```python
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a  1
b  2
dtype: int64

>>> s_copy = s.copy()
>>> s_copy
a  1
b  2
dtype: int64

Shallow copy versus default (deep) copy:

```python
default
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)

Shallow copy shares data and index with original.

```python
>>> s is shallow False
>>> s.values is shallow.values and s.index is shallow.index True

Deep copy has own copy of data and index.

```python
>>> s is deep False
>>> s.values is deep.values or s.index is deep.index False

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```python
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a  3
b  4
dtype: int64
```

```python
>>> shallow
a  3
b  4
dtype: int64
```

```python
>>> deep
a  1
b  2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0   [10, 2]
1    [3, 4]
dtype: object
>>>
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object

pandas.DataFrame.corr

DataFrame.corr(method='pearson', min_periods=1)
Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method [{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:
• pearson : standard correlation coefficient
• kendall : Kendall Tau correlation coefficient
• spearman : Spearman rank correlation
• callable: callable with input two 1d ndarrays and returning a float. Note
that the returned matrix from corr will have 1 along the diagonals and
will be symmetric regardless of the callable’s behavior.

min_periods [int, optional] Minimum number of observations required per pair of
columns to have a valid result. Currently only available for Pearson and Spearman
correlation.

Returns

DataFrame Correlation matrix.

See also:
DataFrame.corrwith Compute pairwise correlation with another DataFrame or Series.
Series.corr Compute the correlation between two Series.

Examples

>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> df = pd.DataFrame([[.2, .3], [.0, .6], [.6, .0], [.2, .1]],
...                     columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
dogs  cats
dogs  1.0  0.3
cats  0.3  1.0
pandas.DataFrame.corrwith

DataFrame.corrwith(other, axis=0, drop=False, method='pearson')

Compute pairwise correlation.

Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other [DataFrame, Series] Object with which to compute correlations.
axis [{0 or ‘index’, 1 or ‘columns’}, default 0] The axis to use. 0 or ‘index’ to compute column-wise, 1 or ‘columns’ for row-wise.
drop [bool, default False] Drop missing indices from result.
method [{‘pearson’, ‘kendall’, ‘spearman’} or callable] Method of correlation:
  • pearson : standard correlation coefficient
  • kendall : Kendall Tau correlation coefficient
  • spearman : Spearman rank correlation
  • callable: callable with input two 1d ndarrays and returning a float.

Returns

Series Pairwise correlations.

See also:

DataFrame.corr Compute pairwise correlation of columns.

pandas.DataFrame.count

DataFrame.count(axis=0, level=None, numeric_only=False)

Count non-NA cells for each column or row.

The values None, NaN, NaT, and optionally numpy.inf (depending on pandas.options.mode.use_inf_as_na) are considered NA.

Parameters

axis [{0 or ‘index’, 1 or ‘columns’}, default 0] If 0 or ‘index’ counts are generated for each column. If 1 or ‘columns’ counts are generated for each row.
level [int or str, optional] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame. A str specifies the level name.
numeric_only [bool, default False] Include only float, int or boolean data.

Returns

Series or DataFrame For each column/row the number of non-NA/null entries. If level is specified returns a DataFrame.

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.value_counts Count unique combinations of columns.
**DataFrame.shape**  Number of DataFrame rows and columns (including NA elements).

**DataFrame.isna**  Boolean same-sized DataFrame showing places of NA elements.

### Examples

Constructing DataFrame from a dictionary:

```python
>>> df = pd.DataFrame({"Person":
... ["John", "Myla", "Lewis", "John", "Myla"],
... "Age": [24., np.nan, 21., 33, 26],
... "Single": [False, True, True, True, False]})
```

```python
>>> df
Person  Age  Single
0  John  24.0  False
1  Myla  NaN  True
2  Lewis 21.0  True
3  John  33.0  True
4  Myla  26.0  False
```

Notice the uncounted NA values:

```python
>>> df.count()
Person  5
Age     4
Single  5
dtype: int64
```

Counts for each row:

```python
>>> df.count(axis='columns')
0  3
1  2
2  3
3  3
4  3
dtype: int64
```

**pandas.DataFrame.cov**

`DataFrame.cov(min_periods=None, ddof=1)`

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

**Parameters**
pandas: powerful Python data analysis toolkit, Release 1.4.2

**min_periods** [int, optional] Minimum number of observations required per pair of columns to have a valid result.

**ddof** [int, default 1] Delta degrees of freedom. The divisor used in calculations is \( N - \text{ddof} \), where \( N \) represents the number of elements.

New in version 1.1.0.

Returns

**DataFrame** The covariance matrix of the series of the DataFrame.

See also:

*Series.cov* Compute covariance with another Series.

*core.window.ExponentialMovingWindow.cov* Exponential weighted sample covariance.

*core.window.Expanding.cov* Expanding sample covariance.

*core.window.Rolling.cov* Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame’s time series. The covariance is normalized by \( N - \text{ddof} \).

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```python
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
                      columns=['dogs', 'cats'])
>>> df.cov()
      dogs  cats
    dogs  0.666667 -1.000000
   cats -1.000000  1.666667

>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
                      columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
   a      b       c      d      e
a 0.998438 -0.020161 0.059277 -0.008943 0.014144
b -0.020161 1.059352 -0.008543 -0.024738 0.009826
c 0.059277 -0.008543 1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486 0.921297 -0.013692
e 0.014144 0.009826 -0.000271 -0.013692 0.977795
```
Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```python
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                    columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
   a     b     c
a  0.316741  NaN -0.150812
b  NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

**pandas.DataFrame.cummax**

Dataframe.cummax(axis=None, skipna=True, *args, **kwargs)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Series or DataFrame** Return cumulative maximum of Series or DataFrame.

See also:

- core.window.Expanding.max Similar functionality but ignores NaN values.
- DataFrame.max Return the maximum over DataFrame axis.
- DataFrame.cummax Return cumulative maximum over DataFrame axis.
- DataFrame.cummin Return cumulative minimum over DataFrame axis.
- DataFrame.cumsum Return cumulative sum over DataFrame axis.
- DataFrame.cumprod Return cumulative product over DataFrame axis.
Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1   NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cummax()
0    2.0
1   NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cummax(skipna=False)
0    2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```

DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0   NaN
2  1.0   0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0   NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`
```python
>>> df.cummax(axis=1)
   A   B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

**pandas.DataFrame.cummin**

DataFrame.cummin(axis=None, skipna=True, *args, **kwargs)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Series or DataFrame** Return cumulative minimum of Series or DataFrame.

See also:

- core.window.Expanding.min Similar functionality but ignores NaN values.
- DataFrame.min Return the minimum over DataFrame axis.
- DataFrame.cummax Return cumulative maximum over DataFrame axis.
- DataFrame.cummin Return cumulative minimum over DataFrame axis.
- DataFrame.cumsum Return cumulative sum over DataFrame axis.
- DataFrame.cumprod Return cumulative product over DataFrame axis.

**Examples**

**Series**

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0  2.0
1  NaN
2  5.0
3  -1.0
4   0.0
dtype: float64
```

By default, NA values are ignored.
```python
>>> s.cummin()
0   2.0
1   NaN
2   2.0
3  -1.0
4  -1.0
dtype: float64

To include NA values in the operation, use skipna=False

```python
>>> s.cummin(skipna=False)
0   2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64

```python
DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
                      ... [3.0, np.nan],
                      ... [1.0, 0.0]],
                      ... columns=list('AB'))

```python
```python
>>> df
    A   B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0

By default, iterates over rows and finds the minimum in each column. This is equivalent to axis=None or axis='index'.

```python
>>> df.cummin()
    A   B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0

To iterate over columns and find the minimum in each row, use axis=1

```python
>>> df.cummin(axis=1)
    A   B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```
DataFrame.cumprod

DataFrame.cumprod(axis=None, skipna=True, *args, **kwargs)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.

skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame Return cumulative product of Series or DataFrame.

See also:

core.window.Expanding.prod Similar functionality but ignores NaN values.

DataFrame.prod Return the product over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])

>>> s
0  2.0
1  NaN
2  5.0
3 -1.0
4  0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0   2.0
1  NaN
2  10.0
3 -10.0
4  -0.0
dtype: float64
```

To include NA values in the operation, use skipna=False
```python
s.cumprod(skipna=False)
0   2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```
Returns

Series or DataFrame  Return cumulative sum of Series or DataFrame.

See also:

core.window.Expanding.sum  Similar functionality but ignores NaN values.

DataFrame.sum  Return the sum over DataFrame axis.

DataFrame.cummax  Return cumulative maximum over DataFrame axis.

DataFrame.cummin  Return cumulative minimum over DataFrame axis.

DataFrame.cumsum  Return cumulative sum over DataFrame axis.

DataFrame.cumprod  Return cumulative product over DataFrame axis.

Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0   2.0
1   NaN
2   5.0
3  -1.0
4   0.0
dtype: float64

By default, NA values are ignored.

>>> s.cumsum()
0   2.0
1   NaN
2   7.0
3   6.0
4   6.0
dtype: float64

To include NA values in the operation, use skipna=False

>>> s.cumsum(skipna=False)
0   2.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```

DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                    columns=list('AB'))
```
By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
   A  B
0  2  1
1  5 NaN
2  6  1
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
   A  B
0  2  3
1  3 NaN
2  1  1
```

**pandas.DataFrame.describe**

```
DataFrame.describe(percentiles=None, include=None, exclude=None, datetime_is_numeric=False)
```

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters**

- `percentiles` [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

- `include` [‘all’, list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for Series. Here are the options:
  - ‘all’ : All columns of the input will be included in the output.
  - A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use ‘category’
  - None (default) : The result will include all numeric columns.

- `exclude` [list-like of dtypes or None (default), optional] A black list of data types to omit from the result. Ignored for Series. Here are the options:
• A list-like of dtypes: Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use 'category'.

• None (default): The result will exclude nothing.

`datetime_is_numeric` [bool, default False] Whether to treat datetime dtypes as numeric. This affects statistics calculated for the column. For DataFrame input, this also controls whether datetime columns are included by default.

New in version 1.1.0.

Returns

**Series or DataFrame** Summary statistics of the Series or Dataframe provided.

See also:

- `DataFrame.count` Count number of non-NA/null observations.
- `DataFrame.max` Maximum of the values in the object.
- `DataFrame.min` Minimum of the values in the object.
- `DataFrame.mean` Mean of the values.
- `DataFrame.std` Standard deviation of the observations.
- `DataFrame.select_dtypes` Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result’s index will include `count, mean, std, min, max` as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result’s index will include `count, unique, top, and freq`. The `top` is the most common value. The `freq` is the most common value’s frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a DataFrame, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a DataFrame are analyzed for the output. The parameters are ignored when analyzing a Series.
Examples

Describing a numeric Series.

```python
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
dtype: float64
```

Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count 4
unique 3
top a
freq 2
dtype: object
```

Describing a timestamp Series.

```python
>>> s = pd.Series([...
...    np.datetime64("2000-01-01"),
...    np.datetime64("2010-01-01"),
...    np.datetime64("2010-01-01")
...])
>>> s.describe(datetime_is_numeric=True)
count 3
mean 2006-09-01 08:00:00
min 2000-01-01 00:00:00
25% 2004-12-31 12:00:00
50% 2010-01-01 00:00:00
75% 2010-01-01 00:00:00
max 2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({
...    'categorical': pd.Categorical(['d', 'e', 'f']),
...    'numeric': [1, 2, 3],
...    'object': ['a', 'b', 'c']
...})
>>> df.describe()
numeric
count 3.0
mean 2.0
std 1.0
(continues on next page)
```
Describing all columns of a DataFrame regardless of data type.

```python
>>> df.describe(include='all')
categorical numeric object
  count     3  3.0   3
  unique    3  NaN   3
  top       f  NaN   a
  freq      1  NaN   1
  mean      NaN  2.0  NaN
  std       NaN  1.0  NaN
  min       NaN  1.0  NaN
  25%       NaN  1.5  NaN
  50%       NaN  2.0  NaN
  75%       NaN  2.5  NaN
  max       NaN  3.0  NaN
```

Describing a column from a DataFrame by accessing it as an attribute.

```python
>>> df.numeric.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```python
>>> df.describe(include=[np.number])
numeric
  count 3.0
  mean 2.0
  std 1.0
  min 1.0
  25% 1.5
  50% 2.0
  75% 2.5
  max 3.0
```

Including only string columns in a DataFrame description.

```python
>>> df.describe(include=[object])
object
  count 3
```
Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])
   categorical
count   3
unique   3
top    d
freq    1
```

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])
   categorical  object
count      3    3
unique     3    3
top        f    a
freq       1    1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[object])
   categorical  numeric
count       3    3.0
unique      3  NaN
mean       NaN   2.0
std        NaN   1.0
min        NaN   1.0
25%        NaN   1.5
50%        NaN   2.0
75%        NaN   2.5
max        NaN   3.0
```

**pandas.DataFrame.diff**

DataFrame.diff(**periods=1, axis=0**)  
First discrete difference of element.  
Calculates the difference of a Dataframe element compared with another element in the Dataframe (default is element in previous row).  

**Parameters**

- **periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Take difference over rows (0) or columns (1).
Returns

**Dataframe**  First differences of the Series.

See also:

**Dataframe.pct_change**  Percent change over given number of periods.

**Dataframe.shift**  Shift index by desired number of periods with an optional time freq.

**Series.diff**  First discrete difference of object.

Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in Dataframe, however dtype of the result is always float64.

Examples

Difference with previous row

```python
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})

>>> df
da b c
0 1 1 1
1 2 1 4
2 3 2 9
3 4 3 16
4 5 5 25
5 6 8 36
```

```python
>>> df.diff()
da b c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0
```

Difference with previous column

```python
>>> df.diff(axis=1)
da b c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0
```

Difference with 3rd previous row

```python
>>> df.diff(3)
da b c
0 NaN 0 0
1 NaN -1 3
2 NaN -1 7
3 NaN -1 13
4 NaN 0 20
5 NaN 2 28
```
```python
>>> df.diff(periods=3)
   a    b    c
0  NaN  NaN  NaN
1  NaN  NaN  NaN
2  NaN  NaN  NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```python
>>> df.diff(periods=-1)
   a    b    c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN  NaN
```

Overflow in input dtype

```python
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
   a
0  NaN
1  255.0
```

**pandas.DataFrame.div**

`DataFrame.div(other, axis='columns', level=None, fill_value=None)`

Get floating division of dataframe and other, element-wise (binary operator `truediv`).

Equivalent to `dataframe / other`, but with support to substitute a `fill_value` for missing data in one of the inputs. With reverse version, `rtruediv`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- `level` [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- `fill_value` [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

`DataFrame` Result of the arithmetic operation.
See also:

*DataFrame.add*  Add DataFrames.
*DataFrame.sub*  Subtract DataFrames.
*DataFrame.mul*  Multiply DataFrames.
*DataFrame.div*  Divide DataFrames (float division).
*DataFrame.truediv*  Divide DataFrames (float division).
*DataFrame.floordiv*  Divide DataFrames (integer division).
*DataFrame.mod*  Calculate modulo (remainder after division).
*DataFrame.pow*  Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
    angles  degrees
circle    0       360
triangle   3       180
rectangle  4       360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
    angles  degrees
circle    1       361
triangle   4       181
rectangle  5       361
```

```python
>>> df.add(1)
    angles  degrees
circle    1       361
triangle   4       181
rectangle  5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
    angles  degrees
circle    0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```
Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358
```

```python
>>> df.sub([1, 2], axis='columns')
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
angles  degrees
circle  -1  359
triangle  2  179
rectangle  3  359
```

Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4],
                      index=['circle', 'triangle', 'rectangle'])
```

```python
>>> other
   angles
circle   0
triangle  3
rectangle 4
```

```python
>>> df * other
angles  degrees
circle   0   NaN
triangle  9   NaN
rectangle 16   NaN
```

```python
>>> df.mul(other, fill_value=0)
   angles  degrees
circle   0   0.0
triangle  9   0.0
rectangle 16   0.0
```

Divide by a MultiIndex by level.
```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
...                             'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                   ['circle', 'triangle', 'rectangle',
...                                    'square', 'pentagon', 'hexagon']])
```

```python
>>> df_multindex
   angles  degrees
A    circle    0    360
    triangle   3    180
    rectangle  4    360
B    square    4    360
    pentagon   5    540
    hexagon   6    720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
A    circle    NaN     1.0
    triangle    1.0     1.0
    rectangle   1.0     1.0
B    square    0.0     0.0
    pentagon   0.0     0.0
    hexagon   0.0     0.0
```

**pandas.DataFrame.divide**

Dataframe.divide(other, axis='columns', level=None, fill_value=None)

Get Floating division of dataframe and other, element-wise (binary operator truediv).

Equivalent to dataframe / other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rtruediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [(0 or ‘index’, 1 or ‘columns’)] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.

**See also:**

- **DataFrame.add** Add DataFrames.
**DataFrame.sub**  Subtract DataFrames.

**DataFrame.mul**  Multiply DataFrames.

**DataFrame.div**  Divide DataFrames (float division).

**DataFrame.truediv**  Divide DataFrames (float division).

**DataFrame.floordiv**  Divide DataFrames (integer division).

**DataFrame.mod**  Calculate modulo (remainder after division).

**DataFrame.pow**  Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])

>>> df
   angles  degrees
   circle   0       360
   triangle 3       180
   rectangle 4       360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
   angles  degrees
   circle   1       361
   triangle 4       181
   rectangle 5       361
```

```python
>>> df.add(1)
   angles  degrees
   circle   1       361
   triangle 4       181
   rectangle 5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
   angles  degrees
   circle  0.0     36.0
   triangle 0.3    18.0
   rectangle 0.4   36.0
```

```python
>>> df.rdiv(10)
   angles  degrees
   circle  inf     0.027778
```

(continues on next page)
Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
   angles  degrees
   circle    -1     358
   triangle   2     178
   rectangle  3     358
```

```python
>>> df.sub([1, 2], axis='columns')
   angles  degrees
   circle    -1     358
   triangle   2     178
   rectangle  3     358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
   angles  degrees
   circle    -1     359
   triangle   2     179
   rectangle  3     359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
```

```python
>>> df * other
   angles  degrees
   circle    0     NaN
   triangle   9     NaN
   rectangle 16     NaN
```

```python
>>> df.mul(other, fill_value=0)
   angles  degrees
   circle    0     0.0
   triangle   9     0.0
   rectangle 16     0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                              ...
       'degrees': [360, 180, 360, 360, 540, 720],
                              ...
       index=['A', 'A', 'A', 'B', 'B', 'B'],
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```python
>>> df_multindex
   angles  degrees
A  circle   0     360
   triangle  3     180
   rectangle 4     360
B  square   4     360
   pentagon  5     540
   hexagon  6     720

>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
A  circle   NaN    1.0
   triangle  1.0    1.0
   rectangle 1.0    1.0
B  square   0.0    0.0
   pentagon  0.0    0.0
   hexagon  0.0    0.0
```

**pandas.DataFrame.dot**

*DataFrame.dot*(other)

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using *self @ other* in Python >= 3.5.

**Parameters**

- **other** [Series, DataFrame or array-like] The other object to compute the matrix product with.

**Returns**

- **Series or DataFrame** If other is a Series, return the matrix product between self and other as a Series. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

**See also:**

- **Series.dot** Similar method for Series.
Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```python
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0   -4
1    5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```python
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
   0  1
0  1  4
1  2  2
```

Note that the dot method give the same result as @

```python
>>> df @ other
   0  1
0  1  4
1  2  2
```

The dot method works also if other is an np.array.

```python
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
   0  1
0  1  4
1  2  2
```

Note how shuffling of the objects does not change the result.

```python
>>> s2 = s.reindex([1, 0, 2, 3])
>>> df.dot(s2)
   0   -4
   1    5
dtype: int64
```
pandas.DataFrame.drop

pandas.DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level. See the user guide <advanced.shown_levels> for more information about the now unused levels.

Parameters

labels [single label or list-like] Index or column labels to drop. A tuple will be used as a single label and not treated as a list-like.

axis [[0 or ‘index’, 1 or ‘columns’], default 0] Whether to drop labels from the index (0 or ‘index’) or columns (1 or ‘columns’).

index [single label or list-like] Alternative to specifying axis (labels, axis=0 is equivalent to index=labels).

columns [single label or list-like] Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels).

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If False, return a copy. Otherwise, do operation inplace and return None.

errors ['ignore', 'raise'], default ‘raise’] If ‘ignore’, suppress error and only existing labels are dropped.

Returns

DataFrame or None DataFrame without the removed index or column labels or None if inplace=True.

Raises

KeyError If any of the labels is not found in the selected axis.

See also:

DataFrame.loc Label-location based indexer for selection by label.

DataFrame.dropna Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop Return Series with specified index labels removed.
Examples

```python
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B  C  D
0  0  1  2  3
1  4  5  6  7
2  8  9 10 11
```

Drop columns

```python
>>> df.drop(['B', 'C'], axis=1)
   A  D
0  0  3
1  4  7
2  8 11
```

```python
>>> df.drop(columns=['B', 'C'])
   A  D
0  0  3
1  4  7
2  8 11
```

Drop a row by index

```python
>>> df.drop([0, 1])
   A  B  C  D
   2  8  9 10 11
```

Drop columns and/or rows of MultiIndex DataFrame

```python
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                            ['speed', 'weight', 'length']],
...                       codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                    data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                          [250, 150], [1.5, 0.8], [320, 250],
...                          [1, 0.8], [0.3, 0.2]])
>>> df
   big  small
lama  speed   45.0  30.0
      weight  200.0 100.0
      length  1.5   1.0
cow   speed   30.0  20.0
      weight  250.0 150.0
      length  1.5   0.8
falcon speed  320.0 250.0
       weight  1.0   0.8
       length  0.3   0.2
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination 'falcon' and 'weight', which deletes only the corresponding row
```python
>>> df.drop(index=('falcon', 'weight'))
     big    small
  lama  speed  45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
  cow   speed  30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon speed  320.0   250.0
        length  0.3     0.2

>>> df.drop(index='cow', columns='small')
     big
  lama  speed  45.0
        weight  200.0
        length  1.5
falcon speed  320.0
        weight  1.0
        length  0.3

>>> df.drop(index='length', level=1)
     big    small
  lama  speed  45.0    30.0
        weight  200.0   100.0
  cow   speed  30.0    20.0
        weight  250.0   150.0
falcon speed  320.0   250.0
        weight  1.0     0.8
```

**pandas.DataFrame.drop_duplicates**

DataFrame.drop_duplicates( subset=None, keep='first', inplace=False, ignore_index=False)

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

**Parameters**

- **subset** [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.
- **keep** ['first', 'last', False], default 'first'] Determines which duplicates (if any) to keep.
  - first : Drop duplicates except for the first occurrence.
  - last : Drop duplicates except for the last occurrence.
  - False : Drop all duplicates.
- **inplace** [bool, default False] Whether to drop duplicates in place or to return a copy.
- **ignore_index** [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.

**Returns**

- **DataFrame or None** DataFrame with duplicates removed or None if inplace=True.
See also:

**DataFrame.value_counts** Count unique combinations of columns.

### Examples

Consider dataset containing ramen rating.

```python
>>> df = pd.DataFrame(
    ...   {'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
    ...   'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
    ...   'rating': [4, 4, 3.5, 15, 5]
    ... })
>>> df
   brand style rating
0    Yum Yum   cup   4.0
1    Yum Yum   cup   4.0
2    Indomie   cup   3.5
3    Indomie   pack  15.0
4    Indomie   pack   5.0
```

By default, it removes duplicate rows based on all columns.

```python
>>> df.drop_duplicates()
   brand style rating
0    Yum Yum   cup   4.0
2    Indomie   cup   3.5
3    Indomie   pack  15.0
4    Indomie   pack   5.0
```

To remove duplicates on specific column(s), use **subset**.

```python
>>> df.drop_duplicates(subset=['brand'])
   brand style rating
0    Yum Yum   cup   4.0
2    Indomie   cup   3.5
```

To remove duplicates and keep last occurrences, use **keep**.

```python
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
   brand style rating
1    Yum Yum   cup   4.0
2    Indomie   cup   3.5
4    Indomie   pack   5.0
```
DataFrame.droplevel

DataFrame.droplevel(level, axis=0)

Return Series/DataFrame with requested index / column level(s) removed.

Parameters

- **level** [int, str, or list-like] If a string is given, must be the name of a level. If list-like, elements must be names or positional indexes of levels.

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the level(s) is removed:
  - 0 or 'index': remove level(s) in column.
  - 1 or 'columns': remove level(s) in row.

Returns

Series/DataFrame  Series/DataFrame with requested index / column level(s) removed.

Examples

```python
>>> df = pd.DataFrame([  ... [1, 2, 3, 4],  ... [5, 6, 7, 8],  ... [9, 10, 11, 12]  ... ]).set_index([0, 1]).rename_axis(['a', 'b'])

>>> df.droplevel('a')
level_1  c  d
level_2  e  f
   b  2  3  4
   6  7  8
  10 11 12

>>> df.droplevel('level_2', axis=1)
level_1  c  d
   a  1  2  3  4
   5  6  7  8
  9 10 11 12
```
pandas.DataFrame.dropna

DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or ‘index’, 1 or ‘columns’}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or ‘index’ : Drop rows which contain missing values.
- 1, or ‘columns’ : Drop columns which contain missing value.

Changed in version 1.0.0: Pass tuple or list to drop on multiple axes. Only a single axis is allowed.

how [{‘any’, ‘all’}, default ‘any’] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- ‘any’ : If any NA values are present, drop that row or column.
- ‘all’ : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [column label or sequence of labels, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

Returns

DataFrame or None DataFrame with NA entries dropped from it or None if inplace=True.

See also:

DataFrame.isna Indicate missing values.

DataFrame.notna Indicate existing (non-missing) values.

DataFrame.fillna Replace missing values.

Series.dropna Drop missing values.

Index.dropna Drop missing indices.

Examples

```python
def = pd.DataFrame({'name': ['Alfred', 'Batman', 'Catwoman'],
    'toy': [np.nan, 'Batmobile', 'Bullwhip'],
```

```
>>> df
    name   toy   born
0  Alfred   NaN  NaT
1   Batman  Batmobile 1940-04-25
2  Catwoman  Bullwhip   NaT
```
Drop the rows where at least one element is missing.

```python
>>> df.dropna()
   name    toy    born
 0  Alfred  NaN    NaT
 1  Batman Batmobile 1940-04-25
 2  Catwoman  Bullwhip    NaT
```

Drop the columns where at least one element is missing.

```python
>>> df.dropna(axis='columns')
   name
 0  Alfred
 1  Batman
 2  Catwoman
```

Drop the rows where all elements are missing.

```python
>>> df.dropna(how='all')
   name    toy    born
 0  Alfred  NaN    NaT
 1  Batman Batmobile 1940-04-25
 2  Catwoman  Bullwhip    NaT
```

Keep only the rows with at least 2 non-NA values.

```python
>>> df.dropna(thresh=2)
   name    toy    born
 1  Batman Batmobile 1940-04-25
 2  Catwoman  Bullwhip    NaT
```

Define in which columns to look for missing values.

```python
>>> df.dropna(subset=['name', 'toy'])
   name    toy    born
 0  Alfred  NaN    NaT
 1  Batman Batmobile 1940-04-25
 2  Catwoman  Bullwhip    NaT
```

Keep the DataFrame with valid entries in the same variable.

```python
>>> df.dropna(inplace=True)
>>> df
   name    toy    born
 1  Batman Batmobile 1940-04-25
```

**pandas.DataFrame.duplicated**

`DataFrame.duplicated(subset=None, keep='first')`

Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

**Parameters**

- `subset [column label or sequence of labels, optional]` Only consider certain columns for identifying duplicates, by default use all of the columns.
- `keep ['first', 'last', False], default 'first'` Determines which duplicates (if any) to mark.
• **first**: Mark duplicates as True except for the first occurrence.
• **last**: Mark duplicates as True except for the last occurrence.
• **False**: Mark all duplicates as True.

**Returns**

- **Series** Boolean series for each duplicated rows.

**See also:**

- `Index.duplicated` Equivalent method on index.
- `Series.duplicated` Equivalent method on Series.
- `Series.drop_duplicates` Remove duplicate values from Series.
- `DataFrame.drop_duplicates` Remove duplicate values from DataFrame.

**Examples**

Consider dataset containing ramen rating.

```python
def = pd.DataFrame(
    ...
    'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
    ...
    'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
    ...
    'rating': [4, 4, 3.5, 15, 5]
    ...
)
def

<table>
<thead>
<tr>
<th>brand</th>
<th>style</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yum Yum</td>
<td>cup</td>
<td>4.0</td>
</tr>
<tr>
<td>Yum Yum</td>
<td>cup</td>
<td>4.0</td>
</tr>
<tr>
<td>Indomie</td>
<td>cup</td>
<td>3.5</td>
</tr>
<tr>
<td>Indomie</td>
<td>pack</td>
<td>15.0</td>
</tr>
<tr>
<td>Indomie</td>
<td>pack</td>
<td>5.0</td>
</tr>
</tbody>
</table>
```

By default, for each set of duplicated values, the first occurrence is set on False and all others on True.

```python
def.duplicated()
0   False
1   True
2   False
3   False
4   False
dtype: bool
```

By using ‘last’, the last occurrence of each set of duplicated values is set on False and all others on True.

```python
def.duplicated(keep='last')
0   True
1   False
2   False
3   False
4   False
dtype: bool
```

By setting keep on False, all duplicates are True.
```python
>>> df.duplicated(keep=False)
0   True
1   True
2   False
3   False
4   False
dtype: bool
```

To find duplicates on specific column(s), use `subset`.

```python
>>> df.duplicated(subset=['brand'])
0   False
1   True
2   False
3   True
4   True
dtype: bool
```

`pandas.DataFrame.eq`

`DataFrame.eq(other, axis='columns', level=None)`

Get Equal to of dataframe and other, element-wise (binary operator `eq`).

Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison operators.

Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [{0 or `index`, 1 or `columns`}, default `columns`] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- `level` [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- `DataFrame of bool` Result of the comparison.

**See also:**

- `DataFrame.eq` Compare DataFrames for equality elementwise.
- `DataFrame.ne` Compare DataFrames for inequality elementwise.
- `DataFrame.le` Compare DataFrames for less than inequality or equality elementwise.
- `DataFrame.lt` Compare DataFrames for strictly less than inequality elementwise.
- `DataFrame.ge` Compare DataFrames for greater than inequality or equality elementwise.
- `DataFrame.gt` Compare DataFrames for strictly greater than inequality elementwise.
Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```python
>>> df = pd.DataFrame({
    'cost': [250, 150, 100],
    'revenue': [100, 250, 300],
}, index=['A', 'B', 'C'])

>>> df
  cost  revenue
A  250     100
B  150     250
C  100     300

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100
  cost  revenue
A False   True
B False  False
C  True  False
```

```python
>>> df.eq(100)
  cost  revenue
A False   True
B False  False
C  True  False
```

When *other* is a *Series*, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])
  cost  revenue
A  True   True
B  True  False
C False   True
```

Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D'], axis='index'))
  cost  revenue
A  True  False
B  True   True
C  True   True
D  True   True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```python
>>> df == [250, 100]
  cost  revenue
A  True   True
```
Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')
   cost  revenue
A   True     False
B  False      True
C   True     False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
                        index=['A', 'B', 'C', 'D'])
>>> df.gt(other)
   cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                               'revenue': [100, 250, 300, 200, 175, 225],
                               index=['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
                               ['A', 'B', 'C', 'A', 'B', 'C']})
>>> df_multindex
   cost  revenue
Q1  A  250    100
    B  150    250
    C  100    300
Q2  A  150    200
    B  300    175
    C  220    225
```

```python
>>> df.le(df_multindex, level=1)
   cost  revenue
Q1  A  True    True
    B  True    True
    C  True    True
Q2  A  False    True
    B  True    False
    C  True    False
```
pandas.DataFrame.equals

DataFrame.equals(other)

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal.

The row/column index do not need to have the same type, as long as the values are considered equal. Corresponding columns must be of the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

testing.assert_series_equal Raises anAssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

testing.assert_frame_equal Like assert_series_equal, but targets DataFrames.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Examples

```python
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
     1  2
    0  10 20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```python
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
     1  2
    0  10 20
```

```python
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```python
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
```

(continues on next page)
DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```python
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
   1  2
0 10.0 20.0
>>> df.equals(different_data_type)
False
```

### pandas.DataFrame.eval

**DataFrame.eval**(expr, inplace=False, **kwargs)

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows `eval` to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

**Parameters**

- `expr` [str] The expression string to evaluate.
- `inplace` [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.
- `**kwargs` See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

**Returns**

- `ndarray, scalar, pandas object, or None` The result of the evaluation or None if `inplace=True`.

**See also:**

- `DataFrame.query` Evaluates a boolean expression to query the columns of a frame.
- `DataFrame.assign` Can evaluate an expression or function to create new values for a column.
- `eval` Evaluate a Python expression as a string using various backends.
Notes

For more details see the API documentation for `eval()`. For detailed examples see `enhancing performance with eval`.

Examples

```python
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})

>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
4  5  2

>>> df.eval('A + B')
   0   1
0  11
1  10
2  9
3  8
4  7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```python
>>> df.eval('C = A + B')
   A  B  C
0  1 10 11
1  2  8 10
2  3  6  9
3  4  4  8
4  5  2  7

>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
4  5  2
```

Use `inplace=True` to modify the original DataFrame.

```python
>>> df.eval('C = A + B', inplace=True)

>>> df
   A  B  C
0  1 10 11
1  2  8 10
2  3  6  9
3  4  4  8
4  5  2  7
```

Multiple columns can be assigned to using multi-line expressions:
```python
>>> df.eval(
    ...  
    ...  C = A + B,
    ...  D = A - B,
    ... )

A   B   C   D
0   1  10  11  -9
1   2  10  10  -6
2   3   6   9  -3
3   4   8   9   0
4   5   2   7   3
```

**pandas.DataFrame.ewm**

DataFrame.ewm(  
    com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True,  
    ignore_na=False, axis=0, times=None, method='single')

Provide exponentially weighted (EW) calculations.

Exactly one parameter: `com`, `span`, `halflife`, or `alpha` must be provided.

**Parameters**

- **com** [float, optional] Specify decay in terms of center of mass
  \[ \alpha = 1 / (1 + \text{com}) \], for \( \text{com} \geq 0 \).

- **span** [float, optional] Specify decay in terms of span
  \[ \alpha = 2 / (\text{span} + 1) \], for \( \text{span} \geq 1 \).

- **halflife** [float, str, timedelta, optional] Specify decay in terms of half-life
  \[ \alpha = 1 - \exp(-\ln(2) / \text{halflife}) \], for \( \text{halflife} > 0 \).

  If `times` is specified, the time unit (str or timedelta) over which an observation decays to half its value. Only applicable to `mean()`, and `halflife` value will not apply to the other functions.

  New in version 1.1.0.

- **alpha** [float, optional] Specify smoothing factor \( \alpha \) directly
  \[ 0 < \alpha \leq 1 \].

- **min_periods** [int, default 0] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

- **adjust** [bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

  - When `adjust=True` (default), the EW function is calculated using weights \( w_t = (1 - \alpha)^t \). For example, the EW moving average of the series \([x_0, x_1, ..., x_t]\) would be:
    \[
    y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + ... + (1 - \alpha)^tx_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 - \alpha)^t}
    \]
• When adjust=False, the exponentially weighted function is calculated recursively:

\[ y_0 = x_0 \]
\[ y_t = (1 - \alpha)y_{t-1} + \alpha x_t, \]

**ignore_na** [bool, default False] Ignore missing values when calculating weights.

• When ignore_na=False (default), weights are based on absolute positions. For example, the weights of \( x_0 \) and \( x_2 \) used in calculating the final weighted average of \([x_0, \text{None}, x_2]\) are \((1 - \alpha)^2\) and 1 if adjust=True, and \((1 - \alpha)^2\) and \(\alpha\) if adjust=False.

• When ignore_na=True, weights are based on relative positions. For example, the weights of \( x_0 \) and \( x_2 \) used in calculating the final weighted average of \([x_0, \text{None}, x_2]\) are \(1 - \alpha\) and 1 if adjust=True, and \(1 - \alpha\) and \(\alpha\) if adjust=False.

**axis** [{0, 1}, default 0] If 0 or 'index', calculate across the rows. If 1 or 'columns', calculate across the columns.

**times** [str, np.ndarray, Series, default None] New in version 1.1.0.

Only applicable to mean().

Times corresponding to the observations. Must be monotonically increasing and datetime64[ns] dtype.

If 1-D array like, a sequence with the same shape as the observations.

Deprecated since version 1.4.0: If str, the name of the column in the DataFrame representing the times.

**method** [str {'single', 'table'}, default 'single'] New in version 1.4.0.

Execute the rolling operation per single column or row ('single') or over the entire object ('table').

This argument is only implemented when specifying engine='numba' in the method call.

Only applicable to mean()

**Returns**

ExponentialMovingWindow subclass

See also:

**rolling** Provides rolling window calculations.

**expanding** Provides expanding transformations.
Notes

See *Windowing Operations* for further usage details and examples.

Examples

```python
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

>>> df.ewm(com=0.5).mean()
   B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

>>> df.ewm(alpha=2 / 3).mean()
   B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

adjust

```python
>>> df.ewm(com=0.5, adjust=True).mean()
   B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213

>>> df.ewm(com=0.5, adjust=False).mean()
   B
0  0.666667
1  1.555556
2  1.555556
3  3.650794
```

ignore_na

```python
>>> df.ewm(com=0.5, ignore_na=True).mean()
   B
0  0.000000
```
times

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```python
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
   B
0  0.000000
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

### pandas.DataFrame.expanding

DataFrame `.expanding(min_periods=1, center=None, axis=0, method='single')`

Provide expanding window calculations.

**Parameters**

- `min_periods` [int, default 1] Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.
- `center` [bool, default False] If False, set the window labels as the right edge of the window index.
  
  If True, set the window labels as the center of the window index.
  
  Deprecated since version 1.1.0.
- `axis` [int or str, default 0] If 0 or 'index', roll across the rows.
  
  If 1 or 'columns', roll across the columns.
- `method` [str {'single', 'table'}, default 'single'] Execute the rolling operation per single column or row ('single') or over the entire object ('table').
  
  This argument is only implemented when specifying `engine='numba'` in the method call.
  
  New in version 1.3.0.

**Returns**

Expanding subclass
See also:

- **rolling** Provides rolling window calculations.
- **ewm** Provides exponential weighted functions.

**Notes**

See *Windowing Operations* for further usage details and examples.

**Examples**

```python
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

**min_periods**

Expanding sum with 1 vs 3 observations needed to calculate a value.

```python
>>> df.expanding(1).sum()
   B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
>>> df.expanding(3).sum()
   B
0  NaN
1  NaN
2  3.0
3  3.0
4  7.0
```

**pandas.DataFrame.explode**

DataFrame.explode(*column*, *ignore_index=False*)

Transform each element of a list-like to a row, replicating index values.

New in version 0.25.0.

**Parameters**

- **column** [IndexLabel] Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

  New in version 1.3.0: Multi-column explode
**ignore_index**  [bool, default False] If True, the resulting index will be labeled 0, 1, ..., n - 1.

New in version 1.1.0.

**Returns**

DataFrame  Exploded lists to rows of the subset columns; index will be duplicated for these rows.

**Raises**

ValueError :

- If columns of the frame are not unique.
- If specified columns to explode is empty list.
- If specified columns to explode have not matching count of elements rowwise in the frame.

**See also:**

DataFrame.unstack  Pivot a level of the (necessarily hierarchical) index labels.

DataFrame.melt  Unpivot a DataFrame from wide format to long format.

Series.explode  Explode a DataFrame from list-like columns to long format.

**Notes**

This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of rows in the output will be non-deterministic when exploding sets.

Reference the user guide for more examples.

**Examples**

```python
>>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
...                     'B': 1,
...                     'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']])
```

>>> df

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 1, 2]</td>
<td>1</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>1</td>
<td>'foo'</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>[]</td>
<td>1</td>
<td>[]</td>
</tr>
<tr>
<td>3</td>
<td>[3, 4]</td>
<td>1</td>
<td>[d, e]</td>
</tr>
</tbody>
</table>

Single-column explode.

```python
>>> df.explode('A')

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>1</td>
<td>'foo'</td>
<td>1</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Multi-column explode.

```python
>>> df.explode(list('AC'))
   A  B  C
0  0  1  a
0  1  1  b
0  2  1  c
1  foo 1  NaN
2  NaN 1  NaN
3  3  1  d
3  4  1  e
```

**pandas.DataFrame.ffill**

`DataFrame.ffill(axis=None, inplace=False, limit=None, downcast=None)`

Synonym for `DataFrame.fillna()` with method='ffill'.

Returns

- Series/DataFrame or None Object with missing values filled or None if inplace=True.

**pandas.DataFrame.fillna**

`DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)`

Fill NA/NaN values using the specified method.

Parameters

- **value** [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

- **method** [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffilt: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.

- **axis** [{0 or 'index', 1 or 'columns'}] Axis along which to fill missing values.

- **inplace** [bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

- **limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.
downcast [dict, default is None] A dict of item->dtype of what to downcast if possible, or the string ‘infer’ which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

DataFrame or None Object with missing values filled or None if inplace=True.

See also:

interpolate Fill NaN values using interpolation.
reindex Conform object to new index.
asfreq Convert TimeSeries to specified frequency.

Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
... [3, 4, np.nan, 1],
... [np.nan, np.nan, np.nan, np.nan],
... [np.nan, 3, np.nan, 4]],
... columns=list("ABCD"))
>>> df
   A  B  C  D
0  NaN 2.0 NaN 0.0
1  3.0 4.0 NaN 1.0
2  NaN NaN NaN NaN
3  NaN 3.0 NaN 4.0

Replace all NaN elements with 0s.
```

```python
>>> df.fillna(0)
   A  B  C  D
0  0.0 2.0 0.0 0.0
1  3.0 4.0 0.0 1.0
2  0.0 0.0 0.0 0.0
3  0.0 3.0 0.0 4.0
```

We can also propagate non-null values forward or backward.

```python
>>> df.fillna(method="ffill")
   A  B  C  D
0  NaN 2.0 NaN 0.0
1  3.0 4.0 NaN 1.0
2  3.0 4.0 NaN 1.0
3  3.0 3.0 NaN 4.0
```

Replace all NaN elements in column ‘A’, ‘B’, ‘C’, and ‘D’, with 0, 1, 2, and 3 respectively.

```python
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
   A  B  C  D
0  0.0 2.0 2.0 0.0
1  3.0 4.0 2.0 1.0
```

(continues on next page)
Only replace the first NaN element.

```python
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0
```

When filling using a DataFrame, replacement happens along the same column names and same indices

```python
>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
   A    B    C    D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  NaN
3  0.0  3.0  0.0  4.0
```

Note that column D is not affected since it is not present in df2.

### pandas.DataFrame.filter

DataFrame.filter(items=None, like=None, regex=None, axis=None)

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters**

- **items** [list-like] Keep labels from axis which are in items.
- **like** [str] Keep labels from axis for which “like in label == True”.
- **regex** [str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.
- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame.

**Returns**

same type as input object

**See also:**

- **DataFrame.loc** Access a group of rows and columns by label(s) or a boolean array.
Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```python
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
                  index=['mouse', 'rabbit'],
                  columns=['one', 'two', 'three'])
>>> df
  one  two  three
mouse  1  2   3
rabbit 4  5   6

>>> # select columns by name
>>> df.filter(items=['one', 'three'])
   one  three
mouse 1   3
rabbit 4   6

>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
   one  three
mouse 1   3
rabbit 4   6

>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
  one  two  three
rabbit 4   5   6
```

`pandas.DataFrame.first`

`DataFrame.first(offset)`

Select initial periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can select the first few rows based on a date offset.

Parameters

- `offset` [str, DateOffset or dateutil.relativedelta] The offset length of the data that will be selected. For instance, ‘1M’ will display all the rows having their index within the first month.

Returns

- `Series` or `DataFrame` A subset of the caller.

Raises

- `TypeError` If the index is not a `DatetimeIndex`
See also:

- **last** Select final periods of time series based on a date offset.
- **at_time** Select values at a particular time of the day.
- **between_time** Select values between particular times of the day.

**Examples**

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
    A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4

Get the rows for the first 3 days:

```python
>>> ts.first('3D')
    A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

**pandas.DataFrame.first_valid_index**

**DataFrame.first_valid_index()**

Return index for first non-NA value or None, if no non-NA value is found.

**Returns**

- **scalar** [type of index]

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

**pandas.DataFrame.floordiv**

**DataFrame.floordiv(other, axis='columns', level=None, fill_value=None)**

Get Integer division of dataframe and other, element-wise (binary operator floordiv).

Equivalent to dataframe // other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rfloordiv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.
other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or ‘index’, 1 or ‘columns’}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360],
...                     index=['circle', 'triangle', 'rectangle'])
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
```

Divide by constant with reverse version.

```python
>>> df.div(10)
angles   degrees
circle   0.0    36.0
triangle 0.3    18.0
rectangle 0.4   36.0
```

```
>>> df.rdiv(10)
angles   degrees
circle   inf   0.027778
triangle 3.333333  0.055556
rectangle 2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
angles   degrees
circle   -1    358
triangle  2    178
rectangle 3    358
```

```
>>> df.sub([1, 2], axis='columns')
angles   degrees
circle   -1    358
triangle  2    178
rectangle 3    358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
angles   degrees
circle   -1    359
triangle  2    179
rectangle 3    359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
```

```
>>> other
angles
circle    0
triangle  3
rectangle 4
```
>>> df * other
    angles  degrees
  circle     0    NaN
  triangle    9    NaN
  rectangle   16    NaN

>>> df.mul(other, fill_value=0)
    angles  degrees
  circle     0     0.0
  triangle    9     0.0
  rectangle   16     0.0

Divide by a MultiIndex by level.

>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                               'degrees': [360, 180, 360, 540, 720],
                               index=['A', 'A', 'A', 'B', 'B'],
                               ['circle', 'triangle', 'rectangle',
                                'square', 'pentagon', 'hexagon'])

>>> df_multindex
     angles  degrees
    A circle     0  360.0
         triangle  3  180.0
          rectangle 4  360.0
     B square     4  360.0
          pentagon  5  540.0
              hexagon  6  720.0

>>> df.div(df_multindex, level=1, fill_value=0)
    angles  degrees
    A circle   NaN   1.0
         triangle  1.0  1.0
          rectangle  1.0  1.0
     B square   0.0   0.0
          pentagon  0.0   0.0
              hexagon  0.0   0.0

**pandas.DataFrame.from_dict**

**classmethod** `DataFrame.from_dict(data, orient='columns', dtype=None, columns=None)`

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

**Parameters**

- **data** [dict] Of the form [field : array-like] or [field : dict].
- **orient** [{‘columns’, ‘index’, ‘tight’}, default ‘columns’] The “orientation” of the data.
  If the keys of the passed dict should be the columns of the resulting DataFrame, pass
  ‘columns’ (default). Otherwise if the keys should be rows, pass ‘index’. If ‘tight’, assume
  a dict with keys [‘index’, ‘columns’, ‘data’, ‘index_names’, ‘column_names’].

  New in version 1.4.0: ‘tight’ as an allowed value for the orient argument.
**dtype** [dtype, default None] Data type to force, otherwise infer.

**columns** [list, default None] Column labels to use when orient='index'. Raises a `ValueError` if used with orient='columns' or orient='tight'.

**Returns**

DataFrame

**See also:**

*DataFrame.from_records* DataFrame from structured ndarray, sequence of tuples or dicts, or `DataFrame`

*DataFrame* DataFrame object creation using constructor.

*DataFrame.to_dict* Convert the DataFrame to a dictionary.

**Examples**

By default the keys of the dict become the DataFrame columns:

```python
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
col_1  col_2
0      a
1      b
2      c
3      d
```

Specify orient='index' to create the DataFrame using dictionary keys as rows:

```python
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
    0  1  2  3
row_1 3  2  1  0
row_2 a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```python
>>> pd.DataFrame.from_dict(data, orient='index', columns=['A', 'B', 'C', 'D'])
    A  B  C  D
row_1 3  2  1  0
row_2 a  b  c  d
```

Specify orient='tight' to create the DataFrame using a 'tight' format:

```python
>>> data = {'index': [('a', 'b'), ('a', 'c')],
...          'columns': [('x', 1), ('y', 2)],
...          'data': [[1, 3], [2, 4]],
...          'index_names': ['n1', 'n2'],
...          'column_names': ['z1', 'z2']}
>>> pd.DataFrame.from_dict(data, orient='tight')
z1  x  y
z2  1  2
```

(continues on next page)
pandas.DataFrame.from_records

classmethod DataFrame.from_records(data, index=None, exclude=None, columns=None, coerce_float=False, nrows=None)

Convert structured or record ndarray to DataFrame.

Creates a DataFrame object from a structured ndarray, sequence of tuples or dicts, or DataFrame.

Parameters

- **data** [structured ndarray, sequence of tuples or dicts, or DataFrame] Structured input data.
- **index** [str, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use.
- **exclude** [sequence, default None] Columns or fields to exclude.
- **columns** [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns).
- **coerce_float** [bool, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.
- **nrows** [int, default None] Number of rows to read if data is an iterator.

Returns

DataFrame

See also:

- **DataFrame.from_dict** DataFrame from dict of array-like or dicts.
- **DataFrame** DataFrame object creation using constructor.

Examples

Data can be provided as a structured ndarray:

```python
>>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
                  dtype=[('col_1', 'i4'), ('col_2', 'U1')])
>>> pd.DataFrame.from_records(data)
  col_1  col_2
0     3     a
1     2     b
2     1     c
3     0     d
```

Data can be provided as a list of dicts:

```python
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data)
  col_1  col_2
0     3     a
1     2     b
2     1     c
3     0     d
```
```python
>>> data = [{'col_1': 3, 'col_2': 'a'},
          {'col_1': 2, 'col_2': 'b'},
          {'col_1': 1, 'col_2': 'c'},
          {'col_1': 0, 'col_2': 'd'}]
>>> pd.DataFrame.from_records(data)
col_1  col_2
0    3    a
1    2    b
2    1    c
3    0    d
```

Data can be provided as a list of tuples with corresponding columns:

```python
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
col_1  col_2
0    3    a
1    2    b
2    1    c
3    0    d
```

### pandas.DataFrame.ge

##### DataFrame.ge(other, axis='columns', level=None)

Get Greater than or equal to of dataframe and other, element-wise (binary operator ge).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ===, !==, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’], default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

**DataFrame of bool** Result of the comparison.

**See also:**

- **DataFrame.eq** Compare DataFrames for equality elementwise.
- **DataFrame.ne** Compare DataFrames for inequality elementwise.
- **DataFrame.le** Compare DataFrames for less than inequality or equality elementwise.
- **DataFrame.lt** Compare DataFrames for strictly less than inequality elementwise.
- **DataFrame.ge** Compare DataFrames for greater than inequality or equality elementwise.
- **DataFrame.gt** Compare DataFrames for strictly greater than inequality elementwise.
Notes

Mismatched indices will be unioned together. \( NaN \) values are considered different (i.e. \( NaN \neq NaN \)).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                     'revenue': [100, 250, 300],
...                     index=['A', 'B', 'C'])
>>> df
      cost  revenue
    A    250      100
    B    150      250
    C    100      300

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100
     cost  revenue
A   False     True
B   False    False
C   True    False
```

```python
>>> df.eq(100)
     cost  revenue
A   False     True
B   False    False
C   True    False
```

When \( other \) is a \texttt{Series}, the columns of a DataFrame are aligned with the index of \( other \) and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])
     cost  revenue
A    True     True
B    True    False
C   False     True

Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D'], axis='index'))
     cost  revenue
A    True    False
B    True     True
C   False     True
D    True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in \( other \):

```python
>>> df == [250, 100]
     cost  revenue
A    True     True
```

(continues on next page)
Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')
cost  revenue
A    True  False
B    False  True
C    True  False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
                        index=['A', 'B', 'C', 'D'])

>>> df.gt(other)
cost  revenue
A    False  False
B    False  False
C    False  True
D    False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                               'revenue': [100, 250, 300, 200, 175, 225],
                               'index':[['Q1', 'Q1', 'Q1', 'Q2', 'Q2'],
                                        ['A', 'B', 'C', 'A', 'B', 'C']]})

>>> df_multindex
   cost  revenue
Q1 A  250     100
     B  150     250
     C  100     300
Q2 A  150     200
     B  300     175
     C  220     225

>>> df.le(df_multindex, level=1)
cost  revenue
Q1 A  True  True
     B  True  True
     C  True  True
Q2 A  False  True
     B  True  False
     C  True  False
```
### pandas.DataFrame.get

**DataFrame.get(key, default=None)**  
Get item from object for given key (ex: DataFrame column).  
Returns default value if not found.

**Parameters**

- **key** [object]

**Returns**

- **value** [same type as items contained in object]

### Examples

```python
>>> df = pd.DataFrame(  
...     [  
...         [24.3, 75.7, "high"],  
...         [31, 87.8, "high"],  
...         [22, 71.6, "medium"],  
...         [35, 95, "medium"],  
...     ],  
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],  
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),  
... )
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>temp_celsius</th>
<th>temp_fahrenheit</th>
<th>windspeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-02-12</td>
<td>24.3</td>
<td>75.7</td>
<td>high</td>
</tr>
<tr>
<td>2014-02-13</td>
<td>31.0</td>
<td>87.8</td>
<td>high</td>
</tr>
<tr>
<td>2014-02-14</td>
<td>22.0</td>
<td>71.6</td>
<td>medium</td>
</tr>
<tr>
<td>2014-02-15</td>
<td>35.0</td>
<td>95.0</td>
<td>medium</td>
</tr>
</tbody>
</table>
```

```python
>>> df.get(["temp_celsius", "windspeed"])  
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>temp_celsius</th>
<th>windspeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-02-12</td>
<td>24.3</td>
<td>high</td>
</tr>
<tr>
<td>2014-02-13</td>
<td>31.0</td>
<td>high</td>
</tr>
<tr>
<td>2014-02-14</td>
<td>22.0</td>
<td>medium</td>
</tr>
<tr>
<td>2014-02-15</td>
<td>35.0</td>
<td>medium</td>
</tr>
</tbody>
</table>
```

If the key isn't found, the default value will be used.

```python
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")  
'default_value'
```
`DataFrame` method `groupby` is used to group DataFrame using a mapper or by a Series of columns. A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

**Parameters**

- **by** [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If by is a function, it’s called on each value of the object’s index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series’ values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Split along rows (0) or columns (1).
- **level** [int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.
- **as_index** [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively “SQL-style” grouped output.
- **sort** [bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.
- **group_keys** [bool, default True] When calling apply, add group keys to index to identify pieces.
- **squeeze** [bool, default False] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.
  
  Deprecated since version 1.1.0.
- **observed** [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.
- **dropna** [bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.
  
  New in version 1.1.0.

**Returns**

`DataFrameGroupBy` Returns a groupby object that contains information about the groups.

**See also:**

- `resample` Convenience method for frequency conversion and resampling of time series.
Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```python
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon', 'Parrot', 'Parrot'],
                   'Max Speed': [380., 370., 24., 26.]})

>>> df
Animal    Max Speed
0  Falcon    380.0
1  Falcon    370.0
2  Parrot    24.0
3  Parrot    26.0

>>> df.groupby(['Animal']).mean()

   Max Speed
Animal
Falcon    375.0
Parrot    25.0

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the `level` parameter:

```python
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
             ['Captive', 'Wild', 'Captive', 'Wild']]

>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))

>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
                    index=index)

>>> df
Animal       Type     Max Speed
Falcon  Captive    390.0
         Wild       350.0
Parrot  Captive    30.0
         Wild       20.0

>>> df.groupby(level=0).mean()

   Max Speed
Animal
Falcon    370.0
Parrot    25.0

>>> df.groupby(level="Type").mean()

   Max Speed
Type
Captive    210.0
Wild       185.0
```

We can also choose to include NA in group keys or not by setting `dropna` parameter, the default setting is `True`.

3.4. DataFrame 1753
```python
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=['a', 'b', 'c'])

>>> df.groupby(by=['b']).sum()
   a  c
  b
1.0 2  3
2.0 2  5

>>> df.groupby(by=['b'], dropna=False).sum()
   a  c
  b
1.0 2  3
2.0 2  5
  NaN 1  4

>>> l = [['a', 12, 12], [None, 12.3, 33.0], ['b', 12.3, 123.0], ['a', 1, 1]]
>>> df = pd.DataFrame(l, columns=['a', 'b', 'c'])

>>> df.groupby(by='a').sum()
   b  c
  a
13.0  13.0
12.3 123.0

>>> df.groupby(by='a', dropna=False).sum()
   b  c
  a
13.0  13.0
12.3 123.0
  NaN 33.0

pandas.DataFrame.gt

DataFrame.gt(other, axis='columns', level=None)
Get Greater than of dataframe and other, element-wise (binary operator gt).
Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.
Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters
- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns
DataFrame of bool  Result of the comparison.

See also:

DataFrame.eq  Compare DataFrames for equality elementwise.
DataFrame.ne  Compare DataFrames for inequality elementwise.
DataFrame.le  Compare DataFrames for less than inequality or equality elementwise.
DataFrame.lt  Compare DataFrames for strictly less than inequality elementwise.
DataFrame.ge  Compare DataFrames for greater than inequality or equality elementwise.
DataFrame.gt  Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300],
...                    index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250     100
B   150     250
C   100     300
```

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100
   cost  revenue
A   False    True
B   False    False
C    True    False
```

```python
>>> df.eq(100)
   cost  revenue
A   False    True
B   False    False
C    True    False
```

When other is a Series, the columns of a DataFrame are aligned with the index of other and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])
   cost  revenue
A    True    True
B    True    False
C   False    True
```

Use the method to control the broadcast axis:
When comparing to an arbitrary sequence, the number of columns must match the number elements in other:

```
>>> df == [250, 100]
cost revenue
A True True
B False False
C False False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
cost revenue
A True False
B False True
C True False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
                        index=['A', 'B', 'C', 'D'])
```

```
>>> df.gt(other)
cost revenue
A False False
B False False
C False True
D False False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                               'revenue': [100, 250, 300, 200, 175, 225],
                               index=['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
                               ...}
```

```
>>> df_multindex
cost revenue
Q1 A 250 100
  B 150 250
  C 100 300
(continues on next page)
```
Q2 A 150 200
B 300 175
C 220 225

```python
>>> df.le(df_multindex, level=1)
   cost  revenue
Q1  A    True   True
    B    True   True
    C    True   True
Q2  A   False   True
    B    True  False
    C    True  False
```

### pandas.DataFrame.head

`DataFrame.head(n=5)`

Return the first $n$ rows.

This function returns the first $n$ rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of $n$, this function returns all rows except the last $n$ rows, equivalent to `df[:-n]`.

**Parameters**

- **n** [int, default 5] Number of rows to select.

**Returns**

- **same type as caller** The first $n$ rows of the caller object.

**See also:**

- `DataFrame.tail` Returns the last $n$ rows.

**Examples**

```python
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion', ...
                           'monkey', 'parrot', 'shark', 'whale', 'zebra']})

>>> df
    animal
 0  alligator
1     bee
2  falcon
3    lion
4  monkey
5   parrot
6   shark
7   whale
8   zebra
```

Viewing the first 5 lines
>>> df.head()
   animal
0  alligator
1      bee
2    falcon
3      lion
4    monkey

Viewing the first \( n \) lines (three in this case)

>>> df.head(3)
   animal
0  alligator
1      bee
2    falcon

For negative values of \( n \)

>>> df.head(-3)
   animal
0  alligator
1      bee
2    falcon
3      lion
4    monkey
5   parrot

pandas.DataFrame.hist

DataFrame.hist(column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None, layout=None, bins=10, backend=None, legend=False, **kwargs)

Make a histogram of the DataFrame’s columns.

A histogram is a representation of the distribution of data. This function calls matplotlib.pyplot.hist(), on each series in the DataFrame, resulting in one histogram per column.

Parameters

data [DataFrame] The pandas object holding the data.

column [str or sequence, optional] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [bool, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.
ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple, optional] The size in inches of the figure to create. Uses the value in matplotlib.rcParams by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend.

New in version 1.0.0.

legend [bool, default False] Whether to show the legend.

New in version 1.1.0.

**kwargs All other plotting keyword arguments to be passed to matplotlib.pyplot.hist().

Returns

matplotlib.AxesSubplot or numpy.ndarray of them

See also:

matplotlib.pyplot.hist Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```python
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
... }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```
DataFrame.idxmax

DataFrame.idxmax(axis=0, skipna=True)

Return index of first occurrence of maximum over requested axis.
NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

Series Indexes of maxima along the specified axis.

Raises

ValueError

- If the row/column is empty

See also:

Series.idxmax Return index of the maximum element.

Notes

This method is the DataFrame version of ndarray.argmax.

Examples

Consider a dataset containing food consumption in Argentina.

```python
>>> df = pd.DataFrame(
    {'consumption': [10.51, 103.11, 55.48],
     'co2_emissions': [37.2, 19.66, 1712]},
    index=['Pork', 'Wheat Products', 'Beef'])
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>consumption</th>
<th>co2_emissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pork</td>
<td>10.51</td>
<td>37.2</td>
</tr>
<tr>
<td>Wheat Products</td>
<td>103.11</td>
<td>19.66</td>
</tr>
<tr>
<td>Beef</td>
<td>55.48</td>
<td>1712.00</td>
</tr>
</tbody>
</table>
```

By default, it returns the index for the maximum value in each column.

```python
>>> df.idxmax()
consumption  Wheat Products
c02_emissions Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis='columns'`.
```python
>>> df.idxmax(axis="columns")
Pork   co2_emissions
Wheat Products consumption
Beef   co2_emissions
dtype: object
```

**pandas.DataFrame.idxmin**

DataFrame.idxmin(axis=0, skipna=True)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

Parameters

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] The axis to use. 0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise.
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

- **Series** Indexes of minima along the specified axis.

Raises

- **ValueError**
  - If the row/column is empty

See also:

- **Series.idxmin** Return index of the minimum element.

Notes

This method is the DataFrame version of ndarray.argmin.

Examples

Consider a dataset containing food consumption in Argentina.

```python
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712],
...                    'index': ['Pork', 'Wheat Products', 'Beef']})
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>consumption</th>
<th>co2_emissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pork</td>
<td>10.51</td>
<td>37.20</td>
</tr>
<tr>
<td>Wheat Products</td>
<td>103.11</td>
<td>19.66</td>
</tr>
<tr>
<td>Beef</td>
<td>55.48</td>
<td>1712.00</td>
</tr>
</tbody>
</table>
```

By default, it returns the index for the minimum value in each column.
To return the index for the minimum value in each row, use `axis="columns"`.

```
>>> df.idxmin(axis="columns")
Pork   consumption
Wheat Products   co2_emissions
Beef   consumption
dtype: object
```

### pandas.DataFrame.infer_objects

`DataFrame.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

**Returns**

converted  [same type as input object]

**See also:**

- `to_datetime` Convert argument to datetime.
- `to_timedelta` Convert argument to timedelta.
- `to_numeric` Convert argument to numeric type.
- `convert_dtypes` Convert argument to best possible dtype.

**Examples**

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
0  1
1  2
2  3

>>> df.dtypes
A   object
dtype: object

>>> df.infer_objects().dtypes
A   int64
dtype: object
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.DataFrame.info

DataFrame.info(verbos=None, buf=None, max_cols=None, memory_usage=None, show_counts=None, null_counts=None)

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

Parameters

- **data** [DataFrame] DataFrame to print information about.
- **verbose** [bool, optional] Whether to print the full summary. By default, the setting in pandas.options.display.max_info_columns is followed.
- **buf** [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output. max_cols : int, optional When to switch from the verbose to the truncated output. If the DataFrame has more than max_cols columns, the truncated output is used. By default, the setting in pandas.options.display.max_info_columns is used.
- **memory_usage** [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the pandas.options.display.memory_usage setting.

  True always show memory usage. False never shows memory usage. A value of ‘deep’ is equivalent to “True with deep introspection”. Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

- **show_counts** [bool, optional] Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than pandas.options.display.max_info_rows and pandas.options.display.max_info_columns. A value of True always shows the counts, and False never shows the counts.

- **null_counts** [bool, optional] Deprecated since version 1.2.0: Use show_counts instead.

Returns

- **None** This method prints a summary of a DataFrame and returns None.

See also:

- DataFrame.describe Generate descriptive statistics of DataFrame columns.
- DataFrame.memory_usage Memory usage of DataFrame columns.
Examples

```python
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({'int_col': int_values, 'text_col': text_values,
...                     'float_col': float_values})
>>> df
   int_col  text_col  float_col
0      1     alpha      0.00
1      2      beta      0.25
2      3     gamma      0.50
3      4     delta      0.75
4      5    epsilon      1.00

Prints information of all columns:

```python
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #     Column    Non-Null Count  Dtype
0     int_col   5 non-null     int64
1     text_col  5 non-null     object
2     float_col 5 non-null    float64
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```python
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```python
>>> import io
... buffer = io.StringIO()
... df.info(buf=buffer)
... s = buffer.getvalue()
... with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
```

The `memory_usage` parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```python
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```python
... 'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...
'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...
'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... }
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
    #   Column      Non-Null Count   Dtype
    ---  ------      --------------   -----  
    0    column_1   1000000 non-null object
    1    column_2   1000000 non-null object
    2    column_3   1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB

>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
    #   Column      Non-Null Count   Dtype
    ---  ------      --------------   -----  
    0    column_1   1000000 non-null object
    1    column_2   1000000 non-null object
    2    column_3   1000000 non-null object
dtypes: object(3)
memory usage: 165.9 MB
```

**pandas.DataFrame.insert**

Dataframe.insert(loc, column, value, allow_duplicates=False)

Insert column into DataFrame at specified location.

Raises a ValueError if column is already contained in the DataFrame, unless allow_duplicates is set to True.

**Parameters**

- `loc` [int] Insertion index. Must verify 0 <= loc <= len(columns).
- `column` [str, number, or hashable object] Label of the inserted column.
- `value` [Scalar, Series, or array-like]
- `allow_duplicates` [bool, optional default False]

**See also:**

- **Index.insert** Insert new item by index.
Examples

```python
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df
   col1  col2
0     1     3
1     2     4

>>> df.insert(1, "newcol", [99, 99])
>>> df
   col1  newcol  col2
0     1       99     3
1     2       99     4

>>> df.insert(0, "col1", [100, 100], allow_duplicates=True)
>>> df
   col1  col1  newcol  col2
0   100     1       99     3
1   100     2       99     4
```

Notice that pandas uses index alignment in case of value from type `Series`:

```python
>>> df.insert(0, "col0", pd.Series([5, 6], index=[1, 2]))
>>> df
   col0  col1  newcol  col2
0   NaN   100     1       99     3
1   5.0   100     2       99     4
```

**pandas.DataFrame.interpolate**

`DataFrame.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)`

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

**Parameters**

- **method** [str, default ‘linear’] Interpolation technique to use. One of:
  - ‘linear’: Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
  - ‘time’: Works on daily and higher resolution data to interpolate given length of interval.
  - ‘index’, ‘values’: use the actual numerical values of the index.
  - ‘pad’: Fill in NaNs using existing values.
  - ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘spline’, ‘barycentric’, ‘polynomial’: Passed to `scipy.interpolate.interp1d`. These methods use the numerical values of the index. Both ‘polynomial’ and ‘spline’ require that you also specify an `order` (int), e.g. `df.interpolate(method='polynomial', order=5)`.
From derivatives': Refers to `scipy.interpolate.BPoly.from_derivatives` which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

**axis** [{0 or 'index', 1 or 'columns', None}], default None Axis to interpolate along.

**limit** [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

**inplace** [bool, default False] Update the data in place if possible.

**limit_direction** [{‘forward’, ‘backward’, ‘both’}], Optional Consecutive NaNs will be filled in this direction.

**If limit is specified:**
- If ‘method’ is ‘pad’ or ‘ffill’, ‘limit_direction’ must be ‘forward’.
- If ‘method’ is ‘backfill’ or ‘bfill’, ‘limit_direction’ must be ‘backwards’.

**If ‘limit’ is not specified:**
- If ‘method’ is ‘backfill’ or ‘bfill’, the default is ‘backward’
- else the default is ‘forward’

Changed in version 1.1.0: raises `ValueError` if `limit_direction` is ‘forward’ or ‘both’ and method is ‘backfill’ or ‘bfill’. raises `ValueError` if `limit_direction` is ‘backward’ or ‘both’ and method is ‘pad’ or ‘ffill’.

**limit_area** [{None, ‘inside’, ‘outside’}], default None If limit is specified, consecutive NaNs will be filled with this restriction.
- None: No fill restriction.
- ‘inside’: Only fill NaNs surrounded by valid values (interpolate).
- ‘outside’: Only fill NaNs outside valid values (extrapolate).

**downcast** [optional, ‘infer’ or None, defaults to None] Downcast dtypes if possible.

**kwargs** [optional] Keyword arguments to pass on to the interpolating function.

**Returns**

- **Series or DataFrame or None** Returns the same object type as the caller, interpolated at some or all NaN values or None if inplace=True.

See also:

- `fillna` Fill missing values using different methods.
- `scipy.interpolate.Akima1DInterpolator` Piecewise cubic polynomials (Akima interpolator).
- `scipy.interpolate.BPoly.from_derivatives` Piecewise polynomial in the Bernstein basis.
- `scipy.interpolate.interp1d` Interpolate a 1-D function.
- `scipy.interpolate.KroghInterpolator` Interpolate polynomial (Krogh interpolator).
- `scipy.interpolate.CubicSpline` Cubic spline data interpolator.
Notes

The ‘krogh’, ‘piecewise_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in NaN in a Series via linear interpolation.

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
```

Filling in NaN in a Series by padding, but filling at most two consecutive NaN at a time.

```python
>>> s = pd.Series([np.nan, "single_one", np.nan, ...
...    "fill_two_more", np.nan, np.nan, np.nan, ...
...    4.71, np.nan])
>>> s
0     NaN
1   single_one
2       NaN
3  fill_two_more
4       NaN
5       NaN
6       NaN
7    4.71
8     NaN
dtype: object
```

Filling in NaN in a Series via polynomial interpolation or splines: Both ‘polynomial’ and ‘spline’ methods require that you also specify an order (int).
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0 0.000000
1 2.000000
2 4.666667
3 8.000000
dtype: float64

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column ‘a’ is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column ‘b’ remains NaN, because there is no entry before it to use for interpolation.

>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
                      (np.nan, 2.0, np.nan, np.nan),
                      (2.0, 3.0, np.nan, 9.0),
                      (np.nan, 4.0, -4.0, 16.0)],
                   columns=list('abcd'))
>>> df
    a         b         c         d
0  0.0       NaN       -1.0       1.0
1  NaN       2.0        NaN       NaN
2  2.0       3.0        NaN       9.0
3  NaN       4.0       -4.0       16.0

Using polynomial interpolation.

>>> df['d'].interpolate(method='polynomial', order=2)
0  1.0
1  4.0
2  9.0
3 16.0
Name: d, dtype: float64

pandas.DataFrame.isin

DataFrame.isin(values)

Whether each element in the DataFrame is contained in values.

Parameters

values [iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If values is a Series, that’s the index. If values is a dict, the keys must be the column names, which must match. If values is a DataFrame, then both the index and column labels must match.

Returns
**DataFrame**  DataFrame of booleans showing whether each element in the DataFrame is contained in values.

**See also:**

- **DataFrame.eq**  Equality test for DataFrame.
- **Series.isin**  Equivalent method on Series.
- **Series.str.contains**  Test if pattern or regex is contained within a string of a Series or Index.

**Examples**

```python
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
                    index=['falcon', 'dog'])
>>> df
   num_legs  num_wings
falcon     2         2
dog        4          0
```

When `values` is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```python
>>> df.isin([0, 2])
   num_legs  num_wings
falcon   True     True
dog     False     True
```

To check if `values` is *not* in the DataFrame, use the `~` operator:

```python
>>> ~df.isin([0, 2])
   num_legs  num_wings
falcon   False    False
dog     True      False
```

When `values` is a dict, we can pass values to check for each column separately:

```python
>>> df.isin({'num_wings': [0, 3]})
   num_legs  num_wings
falcon   False    False
dog     False     True
```

When `values` is a Series or DataFrame the index and column must match. Note that ‘falcon’ does not match based on the number of legs in other.

```python
>>> other = pd.DataFrame({'num_legs': [8, 3], 'num_wings': [0, 2]},
                        index=['spider', 'falcon'])
>>> df.isin(other)
   num_legs  num_wings
falcon   False     True
dog     False    False
```
pandas.DataFrame.isna

DataFrame.isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or \texttt{numpy.NaN}, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or \texttt{numpy.inf} are not considered NA values (unless you set \texttt{pandas.options.mode.use_inf_as_na = True}).

Returns

\textbf{DataFrame} Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

\textbf{DataFrame.isnull} Alias of isna.

\textbf{DataFrame.notna} Boolean inverse of isna.

\textbf{DataFrame.dropna} Omit axes labels with missing values.

\textit{isna} Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
... born=[pd.NaT, pd.Timestamp('1939-05-27'),
... pd.Timestamp('1940-04-25')],
... name=['Alfred', 'Batman', ''],
... toy=[None, 'Batmobile', 'Joker']))
```

```
>>> df
   age    born          name  toy
0  5.0    NaT      Alfred    None
1  6.0 1939-05-27    Batman  Batmobile
2 NaN  1940-04-25     Joker
```

```
>>> df.isna()
   age   born   name   toy
0  False  True  False  True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])
```

```
>>> ser
0    5.0
1    6.0
2  NaN
dtype: float64
```
pandas.DataFrame.isnull

DataFrame.isnull()

DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.Nan, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

DataFrame  Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

See also:

DataFrame.isnull  Alias of isna.
DataFrame.notna  Boolean inverse of isna.
DataFrame.dropna  Omit axes labels with missing values.
isna  Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                        pd.Timestamp('1940-04-25')],
...                    name=['Alfred', 'Batman', ''],
...                    toy=[None, 'Batmobile', 'Joker']))
>>> df
     age  born          name  toy
0    5.0  NaT    Alfred     None
1    6.0 1939-05-27  Batman  Batmobile
2  NaN  1940-04-25  Batman       Joker
```

```python
>>> df.isna()
     age  born  name  toy
0   False  True False  True
1   False  False False  False
2    True  False False  False
```

Show which entries in a Series are NA.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2   NaN
dtype: float64
```

```python
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

**pandas.DataFrame.items**

DataFrame.items() Iterate over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

- **label** [object] The column names for the DataFrame being iterated over.
- **content** [Series] The column entries belonging to each label, as a Series.

See also:

- DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.
- DataFrame.itertuples Iterate over DataFrame rows as namedtuples of the values.

**Examples**

```python
>>> df = pd.DataFrame({‘species’: [‘bear’, ‘bear’, ‘marsupial’],
... ‘population’: [1864, 22000, 80000]},
... index=[‘panda’, ‘polar’, ‘koala’])
>>> df
   species population
panda  bear    1864
polar  bear   22000
koala  marsupial  80000
```

```python
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep=\n')
...     print(f'\nName: species, dtype: object')
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

<table>
<thead>
<tr>
<th>content:</th>
<th>panda 1864</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>polar 22000</td>
</tr>
<tr>
<td></td>
<td>koala 80000</td>
</tr>
<tr>
<td>Name:</td>
<td>population, dtype: int64</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.iteritems**

`DataFrame.iteritems()` iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

**Yields**

- **label** [object] The column names for the DataFrame being iterated over.
- **content** [Series] The column entries belonging to each label, as a Series.

**Examples**

```python
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
                      'population': [1864, 22000, 80000],
                      'index': ['panda', 'polar', 'koala'])
```

```python
>>> df
species  population
panda    bear 1864
polar    bear 22000
koala    marsupial 80000
```

```python
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep='
')
```

```
label: species
content:
panda bear
polar bear
koala marsupial
Name: species, dtype: object

label: population
content:
panda 1864
polar 22000
koala 80000
Name: population, dtype: int64
```
pandas.DataFrame.iterrows

DataFrame.iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

Yields

- **index** [label or tuple of label] The index of the row. A tuple for a MultiIndex.
- **data** [Series] The data of the row as a Series.

See also:

- DataFrame.itertuples Iterate over DataFrame rows as namedtuples of the values.
- DataFrame.items Iterate over (column name, Series) pairs.

Notes

1. Because iterrows returns a Series for each row, it does not preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```python
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int 1.0
float 1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use itertuples() which returns namedtuples of the values and which is generally faster than iterrows.

2. You should never modify something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

pandas.DataFrame.itertuples

DataFrame.itertuples(index=True, name='Pandas')

Iterate over DataFrame rows as namedtuples.

Parameters

- **index** [bool, default True] If True, return the index as the first element of the tuple.
- **name** [str or None, default “Pandas”] The name of the returned namedtuples or None to return regular tuples.

Returns

- **iterator** An object to iterate over namedtuples for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

See also:
**DataFrame.iterrows** Iterate over DataFrame rows as (index, Series) pairs.

**DataFrame.items** Iterate over (column name, Series) pairs.

**Notes**

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. On python versions < 3.7 regular tuples are returned for DataFrames with a large number of columns (>254).

**Examples**

```python
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
                   index=['dog', 'hawk'])
>>> df
    num_legs num_wings
  dog       4         0
  hawk      2         2
>>> for row in df.itertuples():
    print(row)
    ...     Pandas(Index='dog', num_legs=4, num_wings=0)
    Pandas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the `index` parameter to False we can remove the index as the first element of the tuple:

```python
>>> for row in df.itertuples(index=False):
    print(row)
    ...     Pandas(num_legs=4, num_wings=0)
    Pandas(num_legs=2, num_wings=2)
```

With the `name` parameter set we set a custom name for the yielded namedtuples:

```python
>>> for row in df.itertuples(name='Animal'):
    print(row)
    ...     Animal(Index='dog', num_legs=4, num_wings=0)
    Animal(Index='hawk', num_legs=2, num_wings=2)
```

**pandas.DataFrame.join**

DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)

Join columns of another DataFrame.

Join columns with `other` DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

**Parameters**

- **other** [DataFrame, Series, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.
on [str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in other, otherwise joins index-on-index. If multiple values given, the other DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how [{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘left’] How to handle the operation of the two objects.

- left: use calling frame’s index (or column if on is specified)
- right: use other’s index.
- outer: form union of calling frame’s index (or column if on is specified) with other’s index, and sort it. lexicographically.
- inner: form intersection of calling frame’s index (or column if on is specified) with other’s index, preserving the order of the calling’s one.
- cross: creates the cartesian product from both frames, preserves the order of the left keys.

New in version 1.2.0.

lsuffix [str, default ‘’] Suffix to use from left frame’s overlapping columns.

rsuffix [str, default ‘’] Suffix to use from right frame’s overlapping columns.

sort [bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

Returns

DataFrame A dataframe containing columns from both the caller and other.

See also:

DataFrame.merge For column(s)-on-column(s) operations.

Notes

Parameters on, lsuffix, and rsuffix are not supported when passing a list of DataFrame objects.

Support for specifying index levels as the on parameter was added in version 0.23.0.

Examples

```python
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
                    ...            'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})

>>> df
   key    A
0    K0  A0
1    K1  A1
2    K2  A2
3    K3  A3
4    K4  A4
5    K5  A5
```
Join DataFrames using their indexes.

```python
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
       key_caller  A  key_other  B
   0      K0      A0   K0   B0
   1      K1      A1   K1   B1
   2      K2      A2   K2   B2
   3      K3      A3   NaN   NaN
   4      K4      A4   NaN   NaN
   5      K5      A5   NaN   NaN
```

If we want to join using the key columns, we need to set key to be the index in both `df` and `other`. The joined DataFrame will have key as its index.

```python
>>> df.set_index('key').join(other.set_index('key'))
   A   B
key
K0  A0  B0
K1  A1  B1
K2  A2  B2
K3  A3   NaN
K4  A4   NaN
K5  A5   NaN
```

Another option to join using the key columns is to use the `on` parameter. DataFrame.join always uses `other`’s index but we can use any column in `df`. This method preserves the original DataFrame’s index in the result.

```python
>>> df.join(other.set_index('key'), on='key')
   key  A   B
   0  K0  A0  B0
   1  K1  A1  B1
   2  K2  A2  B2
   3  K3  A3   NaN
   4  K4  A4   NaN
   5  K5  A5   NaN
```

Using non-unique key values shows how they are matched.

```python
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'], ...
     'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```python
>>> df
   key  A
   0  K0  A0
   1  K1  A1
   2  K2  A2
   3  K3  A3
   4  K4  A4
   5  K5  A5
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```python
>>> df.join(other.set_index('key'), on='key')
    key  A   B
   0  K0  A0  B0
   1  K1  A1  B1
   2  K1  A2  B1
   3  K3  A3  NaN
   4  K0  A4  B0
   5  K1  A5  B1
```

**pandas.DataFrame.keys**

*DataFrame.keys()*

Get the ‘info axis’ (see Indexing for more).

This is index for Series, columns for DataFrame.

**Returns**

Index Info axis.

**pandas.DataFrame.kurt**

*DataFrame.kurt*(axis=NoDefault.no_default, skipna=True, level=No, numeric_only=No, **kwargs)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

**Parameters**

axis  [[index (0), columns (1)]] Axis for the function to be applied on.

skipna  [bool, default True] Exclude NA/null values when computing the result.

level  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

**Returns**

Series or DataFrame (if level specified)
pandas.DataFrame.kurtosis

DataFrame.kurtosis(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return unbiased kurtosis over requested axis.
Kurtosis obtained using Fisher’s definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [[index (0), columns (1)]] Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)

pandas.DataFrame.last

DataFrame.last(offset)

Select final periods of time series data based on a date offset.
For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters

offset [str, DateOffset, dateutil.relativedelta] The offset length of the data that will be selected. For instance, ‘3D’ will display all the rows having their index within the last 3 days.

Returns

Series or DataFrame A subset of the caller.

Raises

TypeError If the index is not a DatetimeIndex

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.
Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
     A
2018-04-09    1
2018-04-11    2
2018-04-13    3
2018-04-15    4
```

Get the rows for the last 3 days:

```python
>>> ts.last('3D')
     A
2018-04-13    3
2018-04-15    4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

**pandas.DataFrame.last_valid_index**

`DataFrame.last_valid_index()`

Return index for last non-NA value or None, if no non-NA value is found.

**Returns**

scalar [type of index]

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

**pandas.DataFrame.le**

`DataFrame.le(other, axis='columns', level=None)`

Get Less than or equal to of dataframe and other, element-wise (binary operator `le`).

Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison operators.

Equivalent to `==`, `/=`, `<=`, `<`, `>=`, `>` with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- `level` [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
Returns

**DataFrame of bool**  Result of the comparison.

See also:

- `DataFrame.eq`  Compare DataFrames for equality elementwise.
- `DataFrame.ne`  Compare DataFrames for inequality elementwise.
- `DataFrame.le`  Compare DataFrames for less than inequality or equality elementwise.
- `DataFrame.lt`  Compare DataFrames for strictly less than inequality elementwise.
- `DataFrame.ge`  Compare DataFrames for greater than inequality or equality elementwise.
- `DataFrame.gt`  Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```python
def = pd.DataFrame({
    'cost': [250, 150, 100],
    'revenue': [100, 250, 300],
}, index=['A', 'B', 'C'])
def
```

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>250</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
<td>250</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
<td>300</td>
</tr>
</tbody>
</table>

Comparison with a scalar, using either the operator or method:

```python
def == 100
def.eq(100)
```

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

When `other` is a `Series`, the columns of a DataFrame are aligned with the index of `other` and broadcast:

```python
def != pd.Series([100, 250], index=['cost', 'revenue'])
```

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

Use the method to control the broadcast axis:
When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]
     cost revenue
    A    True    True
    B    False   False
    C    False   False
```

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')
     cost revenue
    A    True    False
    B    False   True
    C    True    False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
                        index=['A', 'B', 'C', 'D'])
```

```python
>>> df.gt(other)
     cost revenue
    A    False   False
    B    False   False
    C    False   True
    D    False   False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                               'revenue': [100, 250, 300, 200, 175, 225],
                               ...}
                               index=['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
                               ...}
                               index=['A', 'B', 'C', 'A', 'B', 'C'])
```

(continues on next page)
```python
>>> df.le(df_multindex, level=1)
cost  revenue
Q1   A   True  True
     B   True  True
     C   True  True
Q2   A   False True
     B   True False
     C   True False
```

### pandas.DataFrame.lookup

**DataFrame.lookup**(row_labels, col_labels)

Label-based “fancy indexing” function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

Deprecated since version 1.2.0: DataFrame.lookup is deprecated, use DataFrame.melt and DataFrame.loc instead. For further details see Looking up values by index/column labels.

**Parameters**

- **row_labels** [sequence] The row labels to use for lookup.
- **col_labels** [sequence] The column labels to use for lookup.

**Returns**

- numpy.ndarray The found values.

### pandas.DataFrame.lt

**DataFrame.lt**(other, axis='columns', level=None)

Get Less than of dataframe and other, element-wise (binary operator lt).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, /=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- **DataFrame of bool** Result of the comparison.
See also:

 DataFrame.eq Compare DataFrames for equality elementwise.
 DataFrame.ne Compare DataFrames for inequality elementwise.
 DataFrame.le Compare DataFrames for less than inequality or equality elementwise.
 DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.
 DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.
 DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
                     'revenue': [100, 250, 300],
                     'index': ['A', 'B', 'C']})
>>> df
    cost  revenue
A    250      100
B    150      250
C    100      300
Comparison with a scalar, using either the operator or method:

```python
```n
>>> df == 100
    cost  revenue
A    False  True
B    False  False
C    True  False
```n
```python
>>> df.eq(100)
    cost  revenue
A    False  True
B    False  False
C    True  False
```n
When other is a Series, the columns of a DataFrame are aligned with the index of other and broadcast:

```python
```n
```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])
    cost  revenue
A    True  True
B    True  False
C    False  True
```n
Use the method to control the broadcast axis:
When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]
   cost  revenue
  A   True   True
  B   False  False
  C   False  False
```

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')
   cost  revenue
  A   True   False
  B   False  True
  C   True   False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({
    'revenue': [300, 250, 100, 150],
   ...

>>> other
       revenue
  A      300
  B      250
  C      100
  D      150
```

```python
>>> df.gt(other)
   cost  revenue
  A   False  False
  B   False  False
  C   False   True
  D   False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({
    'cost': [250, 150, 100, 150, 300, 220],
    'revenue': [100, 250, 300, 200, 175, 225],
    ...

>>> df_multindex
     cost  revenue
  Q1 A    250      100
      B    150      250
      C    100      300
(continues on next page)```
Q2 | A 150 | 200 |
| B 300 | 175 |
| C 220 | 225 |

```python
>>> df.le(df_multindex, level=1)
     cost  revenue
    Q1  A    True   True
        B    True   True
        C    True   True
    Q2  A    False  True
        B    True  False
        C    True  False
```

`pandas.DataFrame.mad`

`DataFrame.mad(axis=None, skipna=True, level=None)`

Return the mean absolute deviation of the values over the requested axis.

Parameters:

- **axis** [index (0), columns (1)] Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

Returns:

- Series or DataFrame (if level specified)

`pandas.DataFrame.mask`

`DataFrame.mask(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=NoDefault.no_default)`

Replace values where the condition is True.

Parameters:

- **cond** [bool Series/DataFrame, array-like, or callable] Where cond is False, keep the original value. Where True, replace with corresponding value from other. If cond is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn’t check it).
- **other** [scalar, Series/DataFrame, or callable] Entries where cond is True are replaced with corresponding value from other. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn’t check it).
- **inplace** [bool, default False] Whether to perform the operation in place on the data.
- **axis** [int, default None] Alignment axis if needed.
- **level** [int, default None] Alignment level if needed.
errors [str, {‘raise’, ‘ignore’}, default ‘raise’] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

- ‘raise’: allow exceptions to be raised.
- ‘ignore’: suppress exceptions. On error return original object.

try_cast [bool, default None] Try to cast the result back to the input type (if possible).

Deprecation since version 1.3.0: Manually cast back if necessary.

Returns

Same type as caller or None if inplace=True.

See also:

DataFrame.where() Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if cond is False the element is used; otherwise the corresponding element from the DataFrame other is used.

The signature for DataFrame.where() differs from numpy.where(). Roughly df1.where(m, df2) is equivalent to np.where(m, df1, df2).

For further details and examples see the mask documentation in indexing.

Examples

```python
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0   NaN
1   1.0
2   2.0
3   3.0
4   4.0
dtype: float64
>>> s.mask(s > 0)
0  0.0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```
```
```
df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
df
A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
A  B
0  0  -1
1 -2  3
2 -4  -5
3  6  -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

pandas.DataFrame.max

DataFrame.max(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return the maximum of the values over the requested axis.

If you want the index of the maximum, use idxmax. This is the equivalent of the numpy.ndarray method argmax.

Parameters

axis  [[index (0), columns (1)]] Axis for the function to be applied on.

skipna  [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count
along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None,
will attempt to use everything, then use only numeric data. Not implemented for
Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)

See also:

Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.sum Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([
    ...     ['warm', 'warm', 'cold', 'cold'],
    ...     ['dog', 'falcon', 'fish', 'spider']],
    ...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded animal
  warm   dog   4
      falcon   2
  cold   fish   0
      spider   8
Name: legs, dtype: int64

>>> s.max()
8
```
pandas.DataFrame.mean

DataFrame.mean(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)
Return the mean of the values over the requested axis.

Parameters
axis ([index (0), columns (1)]) Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns
Series or DataFrame (if level specified)

pandas.DataFrame.median

DataFrame.median(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)
Return the median of the values over the requested axis.

Parameters
axis ([index (0), columns (1)]) Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns
Series or DataFrame (if level specified)

pandas.DataFrame.melt

DataFrame.melt(id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None, ignore_index=True)
Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (id_vars), while all other columns, considered measured variables (value_vars), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

Parameters
id_vars [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.
value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
var_name [scalar] Name to use for the ‘variable’ column. If None it uses frame.columns.name or ‘variable’.
value_name [scalar, default ‘value’] Name to use for the ‘value’ column.
col_level [int or str, optional] If columns are a MultiIndex then use this level to melt.
ignore_index [bool, default True] If True, original index is ignored. If False, the original index is retained. Index labels will be repeated as necessary.

New in version 1.1.0.

Returns

DataFrame Unpivoted DataFrame.

See also:
melt Identical method.
pivot_table Create a spreadsheet-style pivot table as a DataFrame.
DataFrame.pivot Return reshaped DataFrame organized by given index / column values.
DataFrame.explode Explode a DataFrame from list-like columns to long format.

Notes

Reference the user guide for more examples.

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
                      'B': {0: 1, 1: 3, 2: 5},
                      'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A  variable  value
0  a         B     1
1  b         B     3
2  c         B     5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A  variable  value
0  a         B     1
1  b         B     3
2  c         B     5
(continues on next page)
```
3 a C 2
4 b C 4
5 c C 6

The names of ‘variable’ and ‘value’ columns can be customized:

```python
>>> df.melt(id_vars=['A'], value_vars=['B'],
       ... var_name='myVarname', value_name='myValname')
   variable  value
0      A  myVarname  myValname
0    B  1
1    B  3
2    B  5

Original index values can be kept around:

```python
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
   variable  value
0      A  myVarname  myValname
0    B  1
1    B  3
2    B  5
0    C  2
1    C  4
2    C  6

If you have multi-index columns:

```python
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
        A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6

>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6

>>> df.melt(id_vars=['A', 'D'], value_vars=['B', 'E'])
   (A, D)  variable_0  variable_1  value
0  a  A  B  E  1
1  b  A  B  E  3
2  c  A  B  E  5
```
pandas.DataFrame.memory_usage

DataFrame.memory_usage(index=True, deep=False)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of object dtype.

This value is displayed in DataFrame.info by default. This can be suppressed by setting pandas.options.display.memory_usage to False.

Parameters

  index [bool, default True] Specifies whether to include the memory usage of the DataFrame’s index in returned Series. If index=True, the memory usage of the index is the first item in the output.

  deep [bool, default False] If True, introspect the data deeply by interrogating object dtypes for system-level memory consumption, and include it in the returned values.

Returns

  Series A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

  numpy.ndarray.nbytes Total bytes consumed by the elements of an ndarray.

  Series.memory_usage Bytes consumed by a Series.

  Categorical Memory-efficient array for string values with many repeated values.

  DataFrame.info Concise summary of a DataFrame.

Examples

```python
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000, dtype=t).astype(t))
... for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
    int64  float64  complex128      object  bool
0     1.0     1.0         1.0+0.0j  1 True
1     1.0     1.0         1.0+0.0j  1 True
2     1.0     1.0         1.0+0.0j  1 True
3     1.0     1.0         1.0+0.0j  1 True
4     1.0     1.0         1.0+0.0j  1 True

>>> df.memory_usage()
   Index 128
   int64  40000
   float64  40000
   complex128  80000
   object  40000
   bool  5000
dtype: int64
```
>>> df.memory_usage(index=False)
int64    40000
float64  40000
complex128  80000
object   40000
bool     5000
dtype: int64

The memory footprint of object dtype columns is ignored by default:

```python
>>> df.memory_usage(deep=True)
Index     128
int64    40000
float64  40000
complex128  80000
object   180000
bool     5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```python
>>> df['object'].astype('category').memory_usage(deep=True)
5244
```

```
pandas.DataFrame.merge
```

```python
DataFrame.merge(right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)
```

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

```python
pandas.DataFrame.merge
```

**Warning:** If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

**Parameters**

- `right` [DataFrame or named Series] Object to merge with.
- `how` [{‘left’, ‘right’, ‘outer’, ‘inner’, ‘cross’}, default ‘inner’] Type of merge to be performed.
  - **left**: use only keys from left frame, similar to a SQL left outer join; preserve key order.
  - **right**: use only keys from right frame, similar to a SQL right outer join; preserve key order.
• outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
• inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
• cross: creates the cartesian product from both frames, preserves the order of the left keys.
  New in version 1.2.0.

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, then the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes [list-like, default is (“_x”, “_y”)] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in left and right respectively. Pass a value of None instead of a string to indicate that the column name from left or right should be left as-is, with no suffix. At least one of the values must not be None.

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, default False] If True, adds a column to the output DataFrame called “_merge” with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of “left_only” for observations whose merge key only appears in the left DataFrame, “right_only” for observations whose merge key only appears in the right DataFrame, and “both” if the observation’s merge key is found in both DataFrames.

validate [str, optional] If specified, checks if merge is of specified type.

• “one_to_one” or “1:1”: check if merge keys are unique in both left and right datasets.
• “one_to_many” or “1:m”: check if merge keys are unique in left dataset.
• “many_to_one” or “m:1”: check if merge keys are unique in right dataset.
• “many_to_many” or “m:m”: allowed, but does not result in checks.

Returns

DataFrame A DataFrame of the two merged objects.
See also:

merge_ordered Merge with optional filling/interpolation.
merge_asof Merge on nearest keys.
DataFrame.join Similar method using indices.

Notes

Support for specifying index levels as the on, left_on, and right_on parameters was added in version 0.23.0
Support for merging named Series objects was added in version 0.24.0

Examples

```python
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
                     'value': [1, 2, 3, 5])
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
                     'value': [5, 6, 7, 8]})
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x  rkey  value_y
0  foo       1  foo       5
1  foo       1  foo       8
2  foo       5  foo       5
3  foo       5  foo       8
4  bar       2  bar       6
5  baz       3  baz       7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
            suffixes=('_left', '_right'))
  lkey  value_left  rkey  value_right
0  foo          1  foo          5
1  foo          1  foo          8
2  foo          5  foo          5
3  foo          5  foo          8
```

(continues on next page)
Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

```python
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
>>> df1
   a  b
0  foo  1
1  bar  2
>>> df2
   a  c
0  foo  3
1  baz  4
>>> df1.merge(df2, how='inner', on='a')
   a  b  c
0  foo  1  3
>>> df1.merge(df2, how='left', on='a')
   a  b  c
0  foo  1  3.0
1  bar  2  NaN
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
>>> df1
   left
0  foo
1  bar
>>> df2
   right
0   7
1   8
>>> df1.merge(df2, how='cross')
   left  right
0  foo   7
1  foo   8
2  bar   7
3  bar   8
```

4 bar 2 bar 6
5 baz 3 baz 7
pandas.DataFrame.min

DataFrame.min(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return the minimum of the values over the requested axis.

If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

Parameters

- **axis** [[index (0), columns (1)]] Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

- **Series** or **DataFrame** (if level specified)

See also:

- **Series.sum** Return the sum.
- **Series.min** Return the minimum.
- **Series.max** Return the maximum.
- **Series.idxmin** Return the index of the minimum.
- **Series.idxmax** Return the index of the maximum.
- **DataFrame.sum** Return the sum over the requested axis.
- **DataFrame.min** Return the minimum over the requested axis.
- **DataFrame.max** Return the maximum over the requested axis.
- **DataFrame.idxmin** Return the index of the minimum over the requested axis.
- **DataFrame.idxmax** Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded    animal
          warm    dog    4
          falcon    fish    0
```

(continues on next page)
>>> s.min()
0

pandas.DataFrame.mod

Dataframe.mod(other, axis='columns', level=None, fill_value=None)

Get Modulo of dataframe and other, element-wise (binary operator mod).
Equivalent to dataframe % other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rmod.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

Dataframe Result of the arithmetic operation.

See also:

Dataframe.add Add DataFrames.
Dataframe.sub Subtract DataFrames.
Dataframe.mul Multiply DataFrames.
Dataframe.div Divide DataFrames (float division).
Dataframe.truediv Divide DataFrames (float division).
Dataframe.floordiv Divide DataFrames (integer division).
Dataframe.mod Calculate modulo (remainder after division).
Dataframe.pow Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
   angles  degrees
circle     0       360
triangle   3       180
rectangle  4       360

Add a scalar with operator version which return the same results.

```python
>>> df + 1
   angles  degrees
circle     1       361
triangle   4       181
rectangle  5       361

>>> df.add(1)
   angles  degrees
circle     1       361
triangle   4       181
rectangle  5       361

Divide by constant with reverse version.

```python
>>> df.div(10)
   angles  degrees
circle   0.0     36.0
triangle 0.3     18.0
rectangle 0.4     36.0

>>> df.rdiv(10)
   angles     degrees
circle     inf    0.027778
triangle  3.33333  0.055556
rectangle 2.50000  0.027778

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
   angles  degrees
circle   -1      358
triangle   2      178
rectangle  3      358
```
```
>>> df.sub([1, 2], axis='columns')
angles  degrees
circle   -1   358
triangle   2   178
rectangle   3   358
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
          axis='index')
angles  degrees
circle   -1   359
triangle   2   179
rectangle   3   359
```

Multiply a DataFrame of different shape with operator version.
```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
                       index=['circle', 'triangle', 'rectangle'])
>>> other
angles  
circle   0
triangle   3
rectangle   4
```
```
>>> df * other
angles  degrees
circle   0   NaN
triangle   9   NaN
rectangle  16   NaN
>>> df.mul(other, fill_value=0)
angles  degrees
circle   0   0.0
triangle   9   0.0
rectangle  16   0.0
```

Divide by a MultiIndex by level.
```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                               'degrees': [360, 180, 360, 540, 720],
                               index=['A', 'A', 'A', 'B', 'B', 'B', 'B'],
                               ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
>>> df_multindex
angles  degrees
A circle   0  360
triangle   3  180
rectangle   4  360
B square   4  360
pentagon   5  540
hexagon   6  720
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.div(df_multindex, level=1, fill_value=0)
                 angles  degrees
A  circle    NaN     1.0
    triangle   1.0     1.0
    rectangle  1.0     1.0
B  square    0.0     0.0
    pentagon  0.0     0.0
    hexagon  0.0     0.0
```

pandas.DataFrame.mode

DataFrame.mode(axis=0, numeric_only=False, dropna=True)

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] The axis to iterate over while searching for the mode:
  - 0 or 'index': get mode of each column
  - 1 or 'columns': get mode of each row.

- **numeric_only** [bool, default False] If True, only apply to numeric columns.

- **dropna** [bool, default True] Don’t consider counts of NaN/NaT.

Returns

- **DataFrame** The modes of each column or row.

See also:

- Series.mode Return the highest frequency value in a Series.
- Series.value_counts Return the counts of values in a Series.

Examples

```python
>>> df = pd.DataFrame([
    ('bird', 2, 2),
    ...    ('mammal', 4, np.nan),
    ...    ('arthropod', 8, 0),
    ...    ('bird', 2, np.nan)],
    ...    index=('falcon', 'horse', 'spider', 'ostrich'),
    ...    columns=('species', 'legs', 'wings'))
>>> df
     species legs  wings
falcon    bird   2     2.0
horse     mammal 4  NaN
spider    arthropod 8   0.0
ostrich   bird   2   NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. Because the resulting DataFrame has two rows, the second row of species and legs contains NaN.
```python
>>> df.mode()
   species  legs  wings
0  bird    2.0    0.0
1  NaN  NaN    2.0

Setting dropna=False NaN values are considered and they can be the mode (like for wings).

```python
>>> df.mode(dropna=False)
   species  legs  wings
0  bird    2.0    NaN
1  NaN    NaN    2.0

Setting numeric_only=True, only the mode of numeric columns is computed, and columns of other types are ignored.

```python
>>> df.mode(numeric_only=True)
   legs  wings
0  2.0    0.0
1  NaN    2.0

To compute the mode over columns and not rows, use the axis parameter:

```python
>>> df.mode(axis='columns', numeric_only=True)
   0  1
falcon  2.0  NaN
horse   4.0  NaN
spider  0.0  8.0
ostrich 2.0  NaN
```

**pandas.DataFrame.mul**

DataFrame.mul(other, axis='columns', level=None, fill_value=None)

Get Multiplication of dataframe and other, element-wise (binary operator mul).

Equivalent to dataframe * other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rmul.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.
See also:

- **Dataframe.add** Add DataFrames.
- **Dataframe.sub** Subtract DataFrames.
- **Dataframe.mul** Multiply DataFrames.
- **Dataframe.div** Divide DataFrames (float division).
- **Dataframe.truediv** Divide DataFrames (float division).
- **Dataframe.floordiv** Divide DataFrames (integer division).
- **Dataframe.mod** Calculate modulo (remainder after division).
- **Dataframe.pow** Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
                     'degrees': [360, 180, 360],
                     'index': ['circle', 'triangle', 'rectangle'])
>>> df
   angles  degrees  index
circle     0      360  circle
triangle   3      180  triangle
rectangle  4      360  rectangle
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
   angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

```python
>>> df.add(1)
   angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
   angles  degrees
circle   0.0      36.0
triangle 0.3      18.0
rectangle 0.4  36.0
```
```python
>>> df.rdiv(10)
 angles   degrees
   circle       inf   0.027778
   triangle  3.333333  0.055556
   rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
 angles   degrees
   circle      -1   358
   triangle      2   178
   rectangle     3   358
```

```python
>>> df.sub([1, 2], axis='columns')
 angles   degrees
   circle      -1   358
   triangle      2   178
   rectangle     3   358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
 angles   degrees
   circle      -1   359
   triangle      2   179
   rectangle     3   359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4], ...
 index=['circle', 'triangle', 'rectangle'])

>>> other
 angles
   circle      0
   triangle     3
   rectangle    4
```

```python
>>> df * other
 angles   degrees
   circle      0   NaN
   triangle     9   NaN
   rectangle    16   NaN
```

```python
>>> df.mul(other, fill_value=0)
 angles   degrees
   circle      0   0.0
   triangle     9   0.0
   rectangle    16   0.0
```

Divide by a MultiIndex by level.

3.4. DataFrame
```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
...   'degrees': [360, 180, 360, 540, 720]},
...   index=[['A', 'A', 'A', 'B', 'B'],
...   ['circle', 'triangle', 'rectangle',
...   'square', 'pentagon', 'hexagon']])

>>> df_multindex
   angles  degrees
A  circle          0    360
    triangle        3    180
    rectangle       4    360
B  square          4    360
    pentagon        5    540
    hexagon         6    720

>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
A  circle          NaN   1.0
    triangle       1.0   1.0
    rectangle      1.0   1.0
B  square          0.0   0.0
    pentagon       0.0   0.0
    hexagon        0.0   0.0
```

**pandas.DataFrame.multiply**

DataFrame.multiply(other, axis='columns', level=None, fill_value=None)

Get Multiplication of dataframe and other, element-wise (binary operator mul).

Equivalent to dataframe * other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rmul.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

Dataframe Result of the arithmetic operation.

**See also:**

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes
Mismatched indices will be unioned together.

Examples

```python
def = pd.DataFrame({'angles': [0, 3, 4],
                   'degrees': [360, 180, 360],
                   'index': ['circle', 'triangle', 'rectangle']})
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
```

```plaintext
angles   degrees
circle   1   361
triangle 4   181
rectangle 5   361
```

```python
>>> df.add(1)
```

```plaintext
angles   degrees
circle   1   361
triangle 4   181
rectangle 5   361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
```

```plaintext
angles   degrees
circle  0.0   36.0
triangle 0.3   18.0
rectangle 0.4   36.0
```

```python
>>> df.rdiv(10)
```

```plaintext
angles   degrees
circle   inf   0.027778
```

(continues on next page)
Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
angles     degrees
  circle    -1   358
  triangle   2   178
  rectangle  3   358
```

```python
>>> df.sub([1, 2], axis='columns')
angles     degrees
  circle    -1   358
  triangle   2   178
  rectangle  3   358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
        axis='index')
angles     degrees
  circle    -1   359
  triangle   2   179
  rectangle  3   359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
                        index=['circle', 'triangle', 'rectangle'])
>>> other
angles
  circle    0
  triangle   3
  rectangle  4
```

```python
>>> df * other
angles     degrees
  circle    0     NaN
  triangle   9     NaN
  rectangle  16     NaN
```

```python
>>> df.mul(other, fill_value=0)
angles     degrees
  circle    0     0.0
  triangle    9   0.0
  rectangle   16   0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                               'degrees': [360, 180, 360, 360, 540, 720],
                               'index':['A', 'A', 'A', 'B', 'B', 'B']},
                              index=[['A', 'A', 'A', 'B', 'B', 'B']],
                              columns=[['angles', 'degrees']])
>>> df_multindex
angles     degrees
  A circle    0   360
  B triangle   3   180
```

(continues on next page)
... ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon']])

```python
>>> df_multindex
   angles  degrees
A  circle     0    360
   triangle   3    180
   rectangle  4    360
B  square     4    360
   pentagon   5    540
   hexagon    6    720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
A  circle   NaN     1.0
   triangle  1.0     1.0
   rectangle 1.0     1.0
B  square   0.0     0.0
   pentagon  0.0     0.0
   hexagon  0.0     0.0
```

### pandas.DataFrame.ne

**DataFrame.ne**(other, axis='columns', level=None)

Get Not equal to of dataframe and other, element-wise (binary operator ne).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, =>, > with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- **DataFrame of bool** Result of the comparison.

**See also:**

- **DataFrame.eq** Compare DataFrames for equality elementwise.
- **DataFrame.ne** Compare DataFrames for inequality elementwise.
- **DataFrame.le** Compare DataFrames for less than inequality or equality elementwise.
- **DataFrame.lt** Compare DataFrames for strictly less than inequality elementwise.
- **DataFrame.ge** Compare DataFrames for greater than inequality or equality elementwise.
- **DataFrame.gt** Compare DataFrames for strictly greater than inequality elementwise.
Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
                    ...        'revenue': [100, 250, 300],
                    ...        index=['A', 'B', 'C'])
>>> df
   cost  revenue
A  250     100
B  150     250
C  100     300

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100
   cost  revenue
A  False   True
B  False   False
C   True   False
```  

```python
>>> df.eq(100)
   cost  revenue
A  False   True
B  False   False
C   True   False
```  

When `other` is a *Series*, the columns of a DataFrame are aligned with the index of `other` and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])
   cost  revenue
A   True   True
B   True   False
C  False   True
```

Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D'], axis='index'))
   cost  revenue
A   True  False
B   True   True
C  False   True
D   True   True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]
   cost  revenue
A   True   True
```  

(continues on next page)
Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')
   cost  revenue
A   True  False
B   False  True
C   True  False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
                        index=['A', 'B', 'C', 'D'])
>>> df.gt(other)
   cost  revenue
A   False  False
B   False  False
C   False  True
D   False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                                'revenue': [100, 250, 300, 200, 175, 225]},
                               index=[['Q1', 'Q2', 'Q1', 'Q2', 'Q2', 'Q2'],
                                      ['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
                                      ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df.le(df_multindex, level=1)
   cost  revenue
Q1 A   True  True
     B   True  True
     C   True  True
Q2 A   False  True
     B   True  False
     C   True  False
```
pandas.DataFrame.nlargest

DataFrame.nlargest(n, columns, keep='first')

Return the first n rows ordered by columns in descending order.

Return the first n rows with the largest values in columns, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to df.sort_values(columns, ascending=False).head(n), but more performant.

Parameters

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{‘first’, ‘last’, ‘all’}, default ‘first’] Where there are duplicate values:

• first : prioritize the first occurrence(s)
• last : prioritize the last occurrence(s)
• all : do not drop any duplicates, even it means selecting more than n items.

Returns

DataFrame The first n rows ordered by the given columns in descending order.

See also:

DataFrame.nsmallest Return the first n rows ordered by columns in ascending order.

DataFrame.sort_values Sort DataFrame by the values.

DataFrame.head Return the first n rows without re-ordering.

Notes

This function cannot be used with all column types. For example, when specifying columns with object or category dtypes, TypeError is raised.

Examples

```python
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
... 434000, 337000, 11300, 11300, 11300],
... 'GDP': [1937894, 2583560, 12011, 4520, 12128,
... 17036, 182, 38, 311],
... 'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN',
... 'IS', 'NR', 'TV', 'AI'],
... 'index':['Italy', 'France', 'Malta',
... 'Maldives', 'Brunei', 'Iceland',
... 'Nauru', 'Tuvalu', 'Anguilla'])
```

```python
>>> df
      population   GDP alpha-2
Italy  59000000  1937894   IT
France 65000000  2583560   FR
```

(continues on next page)
In the following example, we will use `nlargest` to select the three rows having the largest values in column “population”.

```python
>>> df.nlargest(3, 'population')
   population    GDP    alpha-2
  France  65000000  2583560   FR
  Italy   59000000  1937894   IT
   Malta   4340000  120111   MT
```

When using `keep='last'`, ties are resolved in reverse order:

```python
>>> df.nlargest(3, 'population', keep='last')
   population    GDP    alpha-2
  France  65000000  2583560   FR
  Italy   59000000  1937894   IT
  Brunei   4340000  12128   BN
```

When using `keep='all'`, all duplicate items are maintained:

```python
>>> df.nlargest(3, 'population', keep='all')
   population    GDP    alpha-2
  France  65000000  2583560   FR
  Italy   59000000  1937894   IT
  Malta   4340000  120111   MT
  Maldives  4340000   4520   MV
  Brunei   4340000  12128   BN
```

To order by the largest values in column “population” and then “GDP”, we can specify multiple columns like in the next example.

```python
>>> df.nlargest(3, ['population', 'GDP'])
   population    GDP    alpha-2
  France  65000000  2583560   FR
  Italy   59000000  1937894   IT
  Brunei   4340000  12128   BN
```
pandas.DataFrame.notna

DataFrame.notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings ‘’ or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull Alias of notna.
DataFrame.isna Boolean inverse of notna.
DataFrame.dropna Omit axes labels with missing values.
notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
... born=[pd.NaT, pd.Timestamp('1939-05-27'),
... pd.Timestamp('1940-04-25')],
... name=['Alfred', 'Batman', ''],
... toy=[None, 'Batmobile', 'Joker']))
>>> df
   age  born       name  toy
0  5.0  NaT      Alfred  None
1  6.0 1939-05-27  Batman  Batmobile
2 NaN 1940-04-25     Joker

>>> df.notna()
   age  born       name  toy
0   True False     True  False
1   True  True     True  True
2  False  True     True  True

Show which entries in a Series are not NA.

>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2   NaN
dtype: float64
pandas.DataFrame.notnull

DataFrame.notnull()  
DataFrame.notnull is an alias for DataFrame.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

DataFrame  Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull  Alias of notna.
DataFrame.isna   Boolean inverse of notna.
DataFrame.dropna  Omit axes labels with missing values.
notna  Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
... born=[pd.NaT, pd.Timestamp('1939-05-27'),
... pd.Timestamp('1940-04-25')],
... name=['Alfred', 'Batman', ''],
... toy=[None, 'Batmobile', 'Joker']))
>>> df
   age  born             name  toy
0  5.0  NaT        Alfred  None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25     Joker
>>> df.notna()
   age  born             name  toy
0  True  False        True  False
1  True   True         True   True
2  False  True         True   True

Show which entries in a Series are not NA.


```python
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64

>>> ser.notna()
0   True
1   True
2  False
dtype: bool
```

### pandas.DataFrame.nsmallest

**DataFrame.nsmallest**(\(n, \text{columns}, \text{keep}='first'\))

Return the first \(n\) rows ordered by \(\text{columns}\) in ascending order.

Return the first \(n\) rows with the smallest values in \(\text{columns}\), in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to \(\text{df.sort_values(\text{columns}, ascending=True).head(n)}\), but more performant.

**Parameters**

- \(n\) [int] Number of items to retrieve.
- \(\text{columns}\) [list or str] Column name or names to order by.
- \(\text{keep}\) [[‘first’, ‘last’, ‘all’], default ‘first’] Where there are duplicate values:
  - ‘first’ : take the first occurrence.
  - ‘last’ : take the last occurrence.
  - ‘all’ : do not drop any duplicates, even it means selecting more than \(n\) items.

**Returns**

DataFrame

**See also:**

- **DataFrame.nlargest** Return the first \(n\) rows ordered by \(\text{columns}\) in descending order.
- **DataFrame.sort_values** Sort DataFrame by the values.
- **DataFrame.head** Return the first \(n\) rows without re-ordering.
Examples

```python
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000, 434000, 337000, 337000, 11300, 11300],
                      'GDP': [1937894, 2583560, 12011, 4520, 12128, 17036, 182, 311],
                      'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN', 'IS', 'NR', 'TV', 'AI'],
                      'index': ['Italy', 'France', 'Malta', 'Maldives', 'Brunei', 'Iceland', 'Nauru', 'Tuvalu', 'Anguilla']})
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column “population”.

```python
>>> df.nsmallest(3, 'population')
```

When using `keep='last'`, ties are resolved in reverse order:

```python
>>> df.nsmallest(3, 'population', keep='last')
```

When using `keep='all'`, all duplicate items are maintained:

```python
>>> df.nsmallest(3, 'population', keep='all')
```

To order by the smallest values in column “population” and then “GDP”, we can specify multiple columns like in the next example.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>> df.nsmallest(3, ['population', 'GDP'])
   population  GDP  alpha-2
Tuvalu        11300  38   TV
Anguilla      11300  311  AI
Nauru         337000 182  NR
```

**pandas.DataFrame.nunique**

DataFrame.nunique(axis=0, dropna=True)

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

**Parameters**

- `axis` [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
- `dropna` [bool, default True] Don't include NaN in the counts.

**Returns**

Series

**See also:**

Series.nunique Method nunique for Series.

DataFrame.count Count non-NA cells for each column or row.

**Examples**

```python
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A  3
B  2
dtype: int64

>>> df.nunique(axis=1)
0  1
1  2
2  2
dtype: int64
```
**pandas.DataFrame.pad**

DataFrame.pad(axis=None, inplace=False, limit=None, downcast=None)

Synonym for DataFrame.fillna() with method='ffill'.

Returns

Series/DataFrame or None Object with missing values filled or None if inplace=True.

**pandas.DataFrame.pct_change**

DataFrame.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)

Percentage change between the current and a prior element. Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

- **periods** [int, default 1] Periods to shift for forming percent change.
- **fill_method** [str, default 'pad'] How to handle NAs before computing percent changes.
- **limit** [int, default None] The number of consecutive NAs to fill before stopping.
- **freq** [DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).
- ****kwargs Additional keyword arguments are passed into DataFrame.shift or Series.shift.

Returns

chg [Series or DataFrame] The same type as the calling object.

See also:

- Series.diff Compute the difference of two elements in a Series.
- DataFrame.diff Compute the difference of two elements in a DataFrame.
- Series.shift Shift the index by some number of periods.
- DataFrame.shift Shift the index by some number of periods.

**Examples**

Series

```python
>>> s = pd.Series([90, 91, 85])
>>> s
0  90
1  91
2  85
dtype: int64
```
See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```python
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64

>>> s.pct_change(fill_method='ffill')
0    NaN
1    NaN
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```python
>>> df = pd.DataFrame({
...    'FR': [4.0405, 4.0963, 4.3149],
...    'GR': [1.7246, 1.7482, 1.8519],
...    'IT': [804.74, 810.01, 860.13],
...    'index': ['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
   FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13

>>> df.pct_change()
   FR   GR   IT
1980-01-01 NaN  NaN  NaN
1980-02-01 0.013810 0.013684 0.006549
1980-03-01 0.053365 0.059318 0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351],
...     index=['GOOG', 'APPL'])
```

```python
>>> df
   2016  2015  2014
GOOG  1769950  1500923  1371819
APPL  30586265  40912316  41403351

>>> df.pct_change(axis='columns', periods=-1)
   2016   2015   2014
GOOG   0.179241 0.094112  NaN
APPL  -0.252395 -0.011860  NaN
```

**pandas.DataFrame.pipe**

**DataFrame.pipe(func, *args, **kwargs)**  
Apply chainable functions that expect Series or DataFrames.

**Parameters**

- **func** [function] Function to apply to the Series/DataFrame. *args* and **kwargs* are passed into func. Alternatively a (callable, data_keyword) tuple where data_keyword is a string indicating the keyword of callable that expects the Series/DataFrame.

- **args** [iterable, optional] Positional arguments passed into func.

- **kwargs** [mapping, optional] A dictionary of keyword arguments passed into func.

**Returns**

- **object** [the return type of func.]

**See also:**

- **DataFrame.apply**  Apply a function along input axis of DataFrame.

- **DataFrame.applymap**  Apply a function elementwise on a whole DataFrame.

- **Series.map**  Apply a mapping correspondence on a Series.

**Notes**

Use .pipe when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```python
>>> func(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```python
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
```

(continues on next page)
If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose $f$ takes its data as $arg2$:

```python
>>> (df.pipe(h)
...      .pipe(g, arg1=a)
...      .pipe((func, 'arg2'), arg1=a, arg3=c)
...    )
```

### pandas.DataFrame.pivot

**DataFrame.pivot**(index=None, columns=None, values=None)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a “pivot” table) based on column values. Uses unique values from specified `index` / `columns` to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

**Parameters**

- **index**: [str or object or a list of str, optional] Column to use to make new frame’s index.
  
  If None, uses existing index.
  
  Changed in version 1.1.0: Also accept list of index names.

- **columns**: [str or object or a list of str] Column to use to make new frame’s columns.
  
  Changed in version 1.1.0: Also accept list of columns names.

- **values**: [str, object or a list of the previous, optional] Column(s) to use for populating new frame’s values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

**Returns**

DataFrame  Returns reshaped DataFrame.

**Raises**

- **ValueError**: When there are any `index`, `columns` combinations with multiple values.

See also:

- **DataFrame.pivot_table**  Generalization of pivot that can handle duplicate values for one index/column pair.

- **DataFrame.unstack**  Pivot based on the index values instead of a column.

- **wide_to_long**  Wide panel to long format. Less flexible but more user-friendly than melt.
Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Reference the user guide for more examples.

Examples

```python
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
                              'two'],
                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
                    'baz': [1, 2, 3, 4, 5, 6],
                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
   foo  bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```python
>>> df.pivot(index='foo', columns='bar', values='baz')
   bar
foo  baz
one  A  B  C
  1  2  3
two 4  5  6
```

```python
>>> df.pivot(index='foo', columns='bar')['baz']
   bar
foo  baz
one  A  B  C
  1  2  3
two 4  5  6
```

```python
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
   baz  zoo
bar  A  B  C  A  B  C
foo  one 1  2  3    x    y    z
     2  3  4  5  6  q  w  t
```

You could also assign a list of column names or a list of index names.

```python
>>> df = pd.DataFrame({
    'lev1': [1, 1, 1, 2, 2, 2],
    'lev2': [1, 1, 2, 1, 1, 2],
    'lev3': [1, 2, 1, 2, 1, 2],
    'lev4': [1, 2, 3, 4, 5, 6],
    'values': [0, 1, 2, 3, 4, 5]})
>>> df
   lev1  lev2  lev3  lev4  values
```

(continues on next page)
0 1 1 1 1 0
1 1 1 2 2 1
2 1 2 1 3 2
3 2 1 2 4 3
4 2 1 1 5 4
5 2 2 2 6 5

```
>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
lev2 1 2
lev3 1 2 1 2
lev1
1 0.0 1.0 2.0 NaN
2 4.0 3.0 NaN 5.0
```

```
>>> df.pivot(index=["lev1", "lev2"], columns=["lev3"], values="values")
lev3 1 2
lev1 lev2
1 1 0.0 1.0
2 2.0 NaN
2 1 4.0 3.0
2 NaN 5.0
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'two', 'two'], ...
                    'bar': ['A', 'A', 'B', 'C'], ...
                    'baz': [1, 2, 3, 4]})
>>> df
   foo bar baz
0   one A   1
1   one A   2
2   two B   3
3   two C   4
```

Notice that the first two rows are the same for our `index` and `columns` arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
...
ValueError: Index contains duplicate entries, cannot reshape
```

```
pandas.DataFrame.pivot_table
```

DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All', observed=False, sort=True)

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

**Parameters**
values  [column to aggregate, optional]

index  [column, Grouper, array, or list of the previous] If an array is passed, it must be
the same length as the data. The list can contain any of the other types (except list).
Keys to group by on the pivot table index. If an array is passed, it is being used as
the same manner as column values.

columns  [column, Grouper, array, or list of the previous] If an array is passed, it must
be the same length as the data. The list can contain any of the other types (except
list). Keys to group by on the pivot table column. If an array is passed, it is being
used as the same manner as column values.

aggfunc  [function, list of functions, dict, default numpy.mean] If list of functions
passed, the resulting pivot table will have hierarchical columns whose top level are
the function names (inferred from the function objects themselves) If dict is passed,
the key is column to aggregate and value is function or list of functions.

fill_value  [scalar, default None] Value to replace missing values with (in the resulting
pivot table, after aggregation).

margins  [bool, default False] Add all row / columns (e.g. for subtotal / grand totals).

dropna  [bool, default True] Do not include columns whose entries are all NaN.

margins_name  [str, default ‘All’] Name of the row / column that will contain the totals
when margins is True.

observed  [bool, default False] This only applies if any of the groupers are Categoricals.
If True: only show observed values for categorical groupers. If False: show all
values for categorical groupers.

Changed in version 0.25.0.

sort  [bool, default True] Specifies if the result should be sorted.

New in version 1.3.0.

Returns

DataFrame  An Excel style pivot table.

See also:

DataFrame.pivot  Pivot without aggregation that can handle non-numeric data.
DataFrame.melt  Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
wide_to_long  Wide panel to long format. Less flexible but more user-friendly than melt.

Notes

Reference the user guide for more examples.
Examples

```python
>>> df = pd.DataFrame({'A': ['foo', 'foo', 'foo', 'foo', 'foo', ... 'bar', 'bar', 'bar', 'bar'], ...
                      'B': ['one', 'one', 'one', 'two', 'two', ...
                           'one', 'one', 'two', 'two'], ...
                      'C': ['small', 'large', 'large', 'small', 'small', ...
                           'large'], ...
                      'D': [1, 2, 2, 3, 3, 4, 5, 6, 7], ...
                      'E': [2, 4, 5, 5, 6, 6, 8, 9, 9]})
```

```text
>>> df
   A    B    C  D  E
0 foo  one small  1  2
1 foo  one large  2  4
2 foo  one large  2  5
3 foo  two small  3  5
4 foo  two small  3  6
5 bar  one large  4  6
6 bar  one small  5  8
7 bar  two small  6  9
8 bar  two large  7  9
```

This first example aggregates values by taking the sum.

```python
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'], ...
                         columns=['C'], aggfunc=np.sum)
```

```text
>>> table
   C       large  small
   A    B
bar one 4.0  5.0
   two 5.0  6.0
foo one 4.0  1.0
   two    NaN  6.0
```

We can also fill missing values using the `fill_value` parameter.

```python
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'], ...
                         columns=['C'], aggfunc=np.sum, fill_value=0)
```

```text
>>> table
   C       large  small
   A    B
bar one 4  5
   two 7  6
foo one 4  1
   two 0  6
```

The next example aggregates by taking the mean across multiple columns.

```python
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'], ...
                         aggfunc={'D': np.mean, ...
                                  'E': np.mean})
```

```text
>>> table
       D     E
   A   B
bar one 4  5
   two 7  6
foo one 4  1
   two 0  6
```

(continues on next page)
We can also calculate multiple types of aggregations for any given value column.

```python
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
                        aggfunc={'D': np.mean,
                                'E': [np.min, np.max, np.mean]})
```

```
<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>large 5.500000 7.500000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>small 5.500000 8.500000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>large 2.000000 4.500000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>small 2.333333 4.333333</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mean max mean min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### pandas.DataFrame.plot

`DataFrame.plot(*args, **kwargs)`

Make plots of Series or DataFrame.

Uses the backend specified by the option `plotting.backend`. By default, matplotlib is used.

**Parameters**

- **data** [Series or DataFrame] The object for which the method is called.
- **x** [label or position, default None] Only used if data is a DataFrame.
- **y** [label, position or list of label, positions, default None] Allows plotting of one column versus another. Only used if data is a DataFrame.
- **kind** [str] The kind of plot to produce:
  - `line`: line plot (default)
  - `bar`: vertical bar plot
  - `barh`: horizontal bar plot
  - `hist`: histogram
  - `box`: boxplot
  - `kde`: Kernel Density Estimation plot
  - `density`: same as `kde`
  - `area`: area plot
  - `pie`: pie plot
  - `scatter`: scatter plot (DataFrame only)
  - `hexbin`: hexbin plot (DataFrame only)
ax [matplotlib axes object, default None] An axes of the current figure.

subplots [bool, default False] Make separate subplots for each column.

sharex [bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in; Be aware, that passing in both an ax and sharex=True will alter all x axis labels for all axis in a figure.

sharey [bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

layout [tuple, optional] (rows, columns) for the layout of subplots.

figsize [a tuple (width, height) in inches] Size of a figure object.

use_index [bool, default True] Use index as ticks for x axis.

title [str or list] Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and subplots is True, print each item in the list above the corresponding subplot.

grid [bool, default None (matlab style default)] Axis grid lines.

legend [bool or {‘reverse’}] Place legend on axis subplots.

style [list or dict] The matplotlib line style per column.

logx [bool or ‘sym’, default False] Use log scaling or symlog scaling on x axis. .. versionchanged:: 0.25.0

logy [bool or ‘sym’ default False] Use log scaling or symlog scaling on y axis. .. versionchanged:: 0.25.0

loglog [bool or ‘sym’, default False] Use log scaling or symlog scaling on both x and y axes. .. versionchanged:: 0.25.0

xticks [sequence] Values to use for the xticks.

yticks [sequence] Values to use for the yticks.

xlim [2-tuple/list] Set the x limits of the current axes.

ylim [2-tuple/list] Set the y limits of the current axes.

xlabel [label, optional] Name to use for the xlabel on x-axis. Default uses index name as xlabel, or the x-column name for planar plots.

    New in version 1.1.0.

    Changed in version 1.2.0: Now applicable to planar plots (scatter, hexbin).

ylabel [label, optional] Name to use for the ylabel on y-axis. Default will show no ylabel, or the y-column name for planar plots.

    New in version 1.1.0.

    Changed in version 1.2.0: Now applicable to planar plots (scatter, hexbin).

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots).

fontsize [int, default None] Font size for xticks and yticks.

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.
**colorbar** [bool, optional] If True, plot colorbar (only relevant for ‘scatter’ and ‘hexbin’ plots).

**position** [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center).

**table** [bool, Series or DataFrame, default False] If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib’s default layout. If a Series or DataFrame is passed, use passed data to draw a table.

**yerr** [DataFrame, Series, array-like, dict and str] See *Plotting with Error Bars* for detail.

**xerr** [DataFrame, Series, array-like, dict and str] Equivalent to yerr.

**stacked** [bool, default False in line and bar plots, and True in area plot] If True, create stacked plot.

**sort_columns** [bool, default False] Sort column names to determine plot ordering.

**secondary_y** [bool or sequence, default False] Whether to plot on the secondary y-axis if a list/tuple, which columns to plot on secondary y-axis.

**mark_right** [bool, default True] When using a secondary_y axis, automatically mark the column labels with “(right)” in the legend.

**include_bool** [bool, default is False] If True, boolean values can be plotted.

**backend** [str, default None] Backend to use instead of the backend specified in the option *plotting.backend*. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set *pd.options.plotting.backend*.

New in version 1.0.0.

**kwargs** Options to pass to matplotlib plotting method.

**Returns**

- matplotlib.axes.Axes or numpy.ndarray of them If the backend is not the default matplotlib one, the return value will be the object returned by the backend.

**Notes**

- See matplotlib documentation online for more on this subject
- If *kind* = ‘bar’ or ‘barh’, you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

---

**pandas.DataFrame.pop**

DataFrame.pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

**Parameters**

- *item* [label] Label of column to be popped.

**Returns**

- Series
Examples

```python
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=('name', 'class', 'max_speed'))
```

```python
>>> df
   name  class  max_speed
0  falcon    bird       389.0
1  parrot    bird        24.0
2    lion  mammal       80.5
3   monkey  mammal       NaN
```

```python
>>> df.pop('class')
0   bird
1   bird
2  mammal
3  mammal
Name: class, dtype: object
```

```python
>>> df
   name  max_speed
0  falcon       389.0
1  parrot        24.0
2    lion       80.5
3   monkey      NaN
```

pandas.DataFrame.pow

DataFrame.pow(other, axis='columns', level=None, fill_value=None)

Get Exponential power of dataframe and other, element-wise (binary operator pow).

Equivalent to dataframe ** other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rpow.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns
**DataFrame**  Result of the arithmetic operation.

**See also:**

- `DataFrame.add`  Add DataFrames.
- `DataFrame.sub`  Subtract DataFrames.
- `DataFrame.mul`  Multiply DataFrames.
- `DataFrame.div`  Divide DataFrames (float division).
- `DataFrame.truediv`  Divide DataFrames (float division).
- `DataFrame.floordiv`  Divide DataFrames (integer division).
- `DataFrame.mod`  Calculate modulo (remainder after division).
- `DataFrame.pow`  Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
                     'degrees': [360, 180, 360],
                     'index': ['circle', 'triangle', 'rectangle']})
>>> df
    angles  degrees
circle     0      360
triangle    3      180
rectangle   4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
    angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

```python
>>> df.add(1)
    angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
    angles  degrees
circle  0.0      36.0
triangle 0.3      18.0
rectangle 0.4      36.0
```
```python
>>> df.rdiv(10)
angles  degrees
    circle  inf  0.027778
    triangle  3.33333  0.055556
    rectangle  2.50000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
angles  degrees
    circle    -1   358
    triangle     2   178
    rectangle     3   358
```

```python
>>> df.sub([1, 2], axis='columns')
angles  degrees
    circle    -1   358
    triangle     2   178
    rectangle     3   358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
angles  degrees
    circle    -1   359
    triangle     2   179
    rectangle     3   359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
                      index=['circle', 'triangle', 'rectangle'])
```

```python
>>> df * other
angles  degrees
    circle        0   NaN
    triangle      9   NaN
    rectangle   16   NaN
```

```python
>>> df.mul(other, fill_value=0)
angles  degrees
    circle        0  0.0
    triangle      9  0.0
    rectangle   16  0.0
```

Divide by a MultiIndex by level.
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
...                              'degrees': [360, 180, 360, 540, 720],
...                              index=[['A', 'A', 'A', 'B', 'B'],
...                              ['circle', 'triangle', 'rectangle',
...                              'square', 'pentagon', 'hexagon']])

>>> df_multindex
       angles  degrees
A circle      0     360
   triangle     3     180
    rectangle    4     360
B   square     4     360
   pentagon     5     540
  hexagon      6     720

>>> df.div(df_multindex, level=1, fill_value=0)
       angles  degrees
A      NaN     1.0
   triangle     1.0     1.0
    rectangle    1.0     1.0
B      0.0     0.0
  pentagon     0.0     0.0
  hexagon     0.0     0.0

pandas.DataFrame.prod

DataFrame.prod(axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs)
Return the product of the values over the requested axis.

Parameters

axis  [[index (0), columns (1)]] Axis for the function to be applied on.

skipna  [bool, default True] Exclude NA/null values when computing the result.

level  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count  [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**kwargs  Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)

See also:

Series.sum  Return the sum.
Series.min  Return the minimum.
Series.max  Return the maximum.
**Series.idxmin**  Return the index of the minimum.

**Series.idxmax**  Return the index of the maximum.

**DataFrame.sum**  Return the sum over the requested axis.

**DataFrame.min**  Return the minimum over the requested axis.

**DataFrame.max**  Return the maximum over the requested axis.

**DataFrame.idxmin**  Return the index of the minimum over the requested axis.

**DataFrame.idxmax**  Return the index of the maximum over the requested axis.

### Examples

By default, the product of an empty or all-NA Series is 1

```python
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```python
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).prod()
1.0
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

### pandas.DataFrame.product

DataFrame.product(*axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs*)

Return the product of the values over the requested axis.

**Parameters**

- **axis**  [[index (0), columns (1)] Axis for the function to be applied on.
- **skipna**  [bool, default True] Exclude NA/null values when computing the result.
- **level**  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only**  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **min_count**  [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.
- ****kwags**  Additional keyword arguments to be passed to the function.

**Returns**
Series or DataFrame (if level specified)

See also:

`Series.sum` Return the sum.
`Series.min` Return the minimum.
`Series.max` Return the maximum.
`Series.idxmin` Return the index of the minimum.
`Series.idxmax` Return the index of the maximum.
`DataFrame.sum` Return the sum over the requested axis.
`DataFrame.min` Return the minimum over the requested axis.
`DataFrame.max` Return the maximum over the requested axis.
`DataFrame.idxmin` Return the index of the minimum over the requested axis.
`DataFrame.idxmax` Return the index of the maximum over the requested axis.

Examples

By default, the product of an empty or all-NA Series is 1

```python
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```python
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).prod()
1.0
```

```python
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

`pandas.DataFrame.quantile`

DataFrame.quantile(q=0.5, axis=0, numeric_only=True, interpolation='linear')

Return values at the given quantile over requested axis.

Parameters

- `q` [float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.
- `axis` [{0, 1, ‘index’, ‘columns’}, default 0] Equals 0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise.
- `numeric_only` [bool, default True] If False, the quantile of datetime and timedelta data will be computed as well.
interpolation  [{‘linear’, ‘lower’, ‘higher’, ‘midpoint’, ‘nearest’}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points i and j:

- linear: \( i + (j - i) \times fraction \), where fraction is the fractional part of the index surrounded by i and j.
- lower: i.
- higher: j.
- nearest: i or j whichever is nearest.
- midpoint: \( (i + j) / 2 \).

Returns

Series or DataFrame

If q is an array, a DataFrame will be returned where the index is q, the columns are the columns of self, and the values are the quantiles.

If q is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

See also:

core.window.Rolling.quantile  Rolling quantile.
numpy.percentile  Numpy function to compute the percentile.

Examples

```python
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                   columns=['a', 'b'])
>>> df.quantile(.1)
a  1.3
b  3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
a  b
0.1 1.3 3.7
0.5 2.5 55.0
```

Specifying numeric_only=False will also compute the quantile of datetime and timedelta data.

```python
>>> df = pd.DataFrame({
    'A': [1, 2],
    'B': [pd.Timestamp('2010'),
           pd.Timestamp('2011')],
    'C': [pd.Timedelta('1 days'),
          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A  1.5
B 2010-07-02 12:00:00
C  1 days 12:00:00
Name: 0.5, dtype: object
```
DataFrame.query

DataFrame.query(expr, inplace=False, **kwargs)
Query the columns of a DataFrame with a boolean expression.

Parameters

expr [str] The query string to evaluate.

You can refer to variables in the environment by prefixing them with an ‘@’ character like @a + b.

You can refer to column names that are not valid Python variable names by surrounding them in backticks. Thus, column names containing spaces or punctuations (besides underscores) or starting with digits must be surrounded by backticks. (For example, a column named ‘Area (cm^2)’ would be referenced as ‘Area (cm^2)’). Column names which are Python keywords (like “list”, “for”, “import”, etc) cannot be used.

For example, if one of your columns is called a a and you want to sum it with b, your query should be ‘a a’ + b.

New in version 0.25.0: Backtick quoting introduced.
New in version 1.0.0: Expanding functionality of backtick quoting for more than only spaces.

inplace [bool] Whether the query should modify the data in place or return a modified copy.

**kwargs See the documentation for eval() for complete details on the keyword arguments accepted by DataFrame.query().

Returns

DataFrame or None DataFrame resulting from the provided query expression or None if inplace=True.

See also:

eval Evaluate a string describing operations on DataFrame columns.

DataFrame.eval Evaluate a string describing operations on DataFrame columns.

Notes

The result of the evaluation of this expression is first passed to DataFrame.loc and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to DataFrame.__getitem__().

This method uses the top-level eval() function to evaluate the passed query.

The query() method uses a slightly modified Python syntax by default. For example, the & and | (bitwise) operators have the precedence of their boolean cousins, and and or. This is syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument parser=’python’. This enforces the same semantics as evaluation in Python space. Likewise, you can pass engine=’python’ to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using numexpr as the engine.
The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the query documentation in `indexing`.

**Backtick quoted variables**

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, `it's` > `that's` will raise an error, as it forms a quoted string (`'s > `that'`) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in pandas.core.computation.parsing.

**Examples**

```python
>>> df = pd.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1))
>>> df
   A  B  C C
0  1 10 10
1  2  8  9
2  3  6  8
3  4  4  7
4  5  2  6
>>> df.query('A > B')
   A  B  C C
4  5  2  6
```

The previous expression is equivalent to

```python
>>> df[df.A > df.B]
   A  B  C C
4  5  2  6
```

For columns with spaces in their name, you can use backtick quoting.

```python
>>> df.query('B == `C C``)
   A  B  C C
0  1 10 10
```

The previous expression is equivalent to
pandas.DataFrame.radd

DataFrame.radd(other, axis='columns', level=None, fill_value=None)

Get Addition of dataframe and other, element-wise (binary operator radd).

Equivalent to other + dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, add.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //. , **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({
    'angles': [0, 3, 4],
    'degrees': [360, 180, 360],
}, index=['circle', 'triangle', 'rectangle'])
```

```
>>> df
          angles  degrees
    circle     0      360
   triangle     3      180
  rectangle     4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
          angles  degrees
    circle     1      361
   triangle     4      181
  rectangle     5      361
```

```python
>>> df.add(1)
          angles  degrees
    circle     1      361
   triangle     4      181
  rectangle     5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
          angles  degrees
    circle   0.0     36.0
   triangle  0.3     18.0
  rectangle  0.4     36.0
```

```python
>>> df.rdiv(10)
          angles  degrees
    circle  inf    0.02778
   triangle 3.33333  0.05556
  rectangle 2.50000  0.02778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
          angles  degrees
    circle    -1      358
   triangle     2      178
  rectangle     3      358
```
>>> df.sub([1, 2], axis='columns')
  angles  degrees
circle   -1      358
triangle  2       178
rectangle 3       358

>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
  angles  degrees
circle   -1      359
triangle  2       179
rectangle 3       359

Multiply a DataFrame of different shape with operator version.

>>> other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])

>>> other
    angles
circle  0
triangle 3
rectangle 4

>>> df * other
  angles  degrees
circle   0      NaN
triangle 9      NaN
rectangle 16     NaN

>>> df.mul(other, fill_value=0)
  angles  degrees
circle   0       0.0
triangle 9       0.0
rectangle 16     0.0

Divide by a MultiIndex by level.

>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                               'degrees': [360, 180, 360, 540, 720],
                               index=['A', 'A', 'A', 'B', 'B', 'B', 'circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])

>>> df_multindex
  angles  degrees
A circle   0       360
  triangle  3       180
  rectangle 4       360
B square   4       360
    pentagon 5       540
    hexagon 6       720
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.div(df_multindex, level=1, fill_value=0)
      angles  degrees
    A  circle    NaN   1.0
         triangle  1.0   1.0
         rectangle  1.0   1.0
    B  square    0.0    0.0
        pentagon  0.0    0.0
        hexagon  0.0    0.0
```

**pandas.DataFrame.rank**

Dataframe.rank(axis=0, method='average', numeric_only=NoDefault.no_default, na_option='keep', ascending=True, pct=False)

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

**Parameters**

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] Index to direct ranking.
- **method** [{‘average’, ‘min’, ‘max’, ‘first’, ‘dense’}, default ‘average’] How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group
  - min: lowest rank in the group
  - max: highest rank in the group
  - first: ranks assigned in order they appear in the array
  - dense: like ‘min’, but rank always increases by 1 between groups.
- **numeric_only** [bool, optional] For DataFrame objects, rank only numeric columns if set to True.
- **na_option** [{‘keep’, ‘top’, ‘bottom’}, default ‘keep’] How to rank NaN values:
  - keep: assign NaN rank to NaN values
  - top: assign lowest rank to NaN values
  - bottom: assign highest rank to NaN values
- **ascending** [bool, default True] Whether or not the elements should be ranked in ascending order.
- **pct** [bool, default False] Whether or not to display the returned rankings in percentile form.

**Returns**

- same type as caller Return a Series or DataFrame with data ranks as values.

**See also:**

- `core.groupby.GroupBy.rank` Rank of values within each group.
Examples

```python
>>> df = pd.DataFrame(data={
    'Animal': ['cat', 'penguin', 'dog',
               'spider', 'snake'],
    'Number_legs': [4, 2, 4, 8, np.nan]})
```

```python
>>> df
Animal  Number_legs
0   cat       4.0
1  penguin   2.0
2    dog       4.0
3  spider     8.0
4   snake    NaN
```

The following example shows how the method behaves with the above parameters:

- **default_rank**: this is the default behaviour obtained without using any parameter.
- **max_rank**: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- **NA_bottom**: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- **pct_rank**: when setting `pct = True`, the ranking is expressed as percentile rank.

```python
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
```

```python
Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0   cat       4.0       2.5         3.0       2.5       0.625
1  penguin   2.0       1.0         1.0       1.0       0.250
2    dog       4.0       2.5         3.0       2.5       0.625
3  spider     8.0       4.0         4.0       4.0       1.000
4   snake    NaN       NaN         NaN       NaN
dataframe
```

**DataFrame.rdiv**

DataFrame.rdiv(other, axis='columns', level=None, fill_value=None)

Get Floating division of dataframe and other, element-wise (binary operator rtruediv).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, truediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
fill_value [float or None, default None] Fill existing missing (NaN) values, and any
new element needed for successful DataFrame alignment, with this value before
computation. If data in both corresponding DataFrame locations is missing the
result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

>>> df = pd.DataFrame({'angles': [0, 3, 4],
... 'degrees': [360, 180, 360]},
...     index=['circle', 'triangle', 'rectangle'])

>>> df
angles  degrees
circle    0     360
triangle   3     180
rectangle  4     360

Add a scalar with operator version which return the same results.

>>> df + 1
angles  degrees
circle    1     361
triangle   4     181
rectangle  5     361

>>> df.add(1)
angles  degrees
circle    1     361
triangle   4     181
rectangle  5     361

Divide by constant with reverse version.
>>> df.div(10)
    angles  degrees
   circle    0.0    36.0
   triangle  0.3     18.0
   rectangle 0.4    36.0

>>> df.rdiv(10)
    angles   degrees
   circle   inf  0.027778
   triangle 3.333333  0.055556
   rectangle 2.500000  0.027778

Subtract a list and Series by axis with operator version.

>>> df - [1, 2]
    angles  degrees
   circle   -1     358
   triangle    2     178
   rectangle    3     358

>>> df.sub([1, 2], axis='columns')
    angles  degrees
   circle   -1     358
   triangle    2     178
   rectangle    3     358

>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
         axis='index')
    angles  degrees
   circle   -1     359
   triangle    2     179
   rectangle    3     359

Multiply a DataFrame of different shape with operator version.

>>> other = pd.DataFrame({'angles': [0, 3, 4],
                        'index': ['circle', 'triangle', 'rectangle']})

>>> df * other
    angles  degrees
   circle    0      NaN
   triangle   9      NaN
   rectangle 16      NaN

>>> df.mul(other, fill_value=0)
    angles  degrees
   circle    0      0.0

(continues on next page)
Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                               'degrees': [360, 180, 360, 360, 540, 720],
                               'index': [['A', 'A', 'A', 'B', 'B', 'B'],
                                         ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon']])
```

```python
>>> df_multindex
angles  degrees
        A     circle  0   360
       triangle  3   180
      rectangle  4   360
        B     square  4   360
       pentagon  5   540
      hexagon  6   720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
angles  degrees
        A     NaN   1.0
       triangle  1.0   1.0
      rectangle  1.0   1.0
        B     square  0.0   0.0
       pentagon  0.0   0.0
      hexagon  0.0   0.0
```

**pandas.DataFrame.reindex**

```python
DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=np.nan, limit=None, tolerance=None)
```

Conform Series/DataFrame to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

**Parameters**

- **keywords for axes** [array-like, optional] New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data.
- **method** [[None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
  - None (default): don’t fill gaps
  - pad / ffill: Propagate last valid observation forward to next valid.
  - backfill / bfill: Use next valid observation to fill gap.
  - nearest: Use nearest valid observations to fill gap.
- **copy** [bool, default True] Return a new object, even if the passed indexes are the same.
level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any “compatible” value.

limit [int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation \( \text{abs(index[indexer] - target)} \leq \text{tolerance} \).

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

Returns

Series/DataFrame with changed index.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

DataFrame.reindex supports two calling conventions

- (index=index_labels, columns=column_labels, ...)
- (labels, axis={"index", "columns"}, ...)

We highly recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```python
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0],
...                     index=index})
>>> df
            http_status  response_time
Firefox        200          0.04
Chrome         200          0.02
Safari         404          0.07
IE10           404          0.08
Konqueror      301          1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

```python
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10', 'Chrome']
```
We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the NaN values.

```python
>>> df.reindex(new_index, fill_value=0)
   http_status  response_time
Safari       404.0         0.07
Iceweasel    NaN           NaN
Comodo Dragon NaN           NaN
IE10         404.0         0.08
Chrome       200.0         0.02
```

We can also reindex the columns.

```python
>>> df.reindex(columns=['http_status', 'user_agent'])
   http_status  user_agent
Firefox     200.0         NaN
Chrome      200.0         NaN
Safari      404.0         NaN
IE10        404.0         NaN
Konqueror   301.0         NaN
```

Or we can use “axis-style” keyword arguments

```python
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
   http_status  user_agent
Firefox     200.0         NaN
Chrome      200.0         NaN
Safari      404.0         NaN
IE10        404.0         NaN
Konqueror   301.0         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```python
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
(continues on next page)
Suppose we decide to expand the dataframe to cover a wider date range.

```python
>>> df2.reindex(date_index2, method='bfill')
prices
2009-12-29 100.0
2009-12-30 100.0
2009-12-31 100.0
2010-01-01 100.0
2010-01-02 101.0
2010-01-03 NaN
2010-01-04 100.0
2010-01-05 89.0
2010-01-06 88.0
2010-01-07 NaN
```

The index entries that did not have a value in the original data frame (for example, ‘2009-12-29’) are by default filled with NaN. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.

```python
>>> df2.reindex(date_index2, method='bfill')
prices
2009-12-29 100.0
2009-12-30 100.0
2009-12-31 100.0
2010-01-01 100.0
2010-01-02 101.0
2010-01-03 NaN
2010-01-04 100.0
2010-01-05 89.0
2010-01-06 88.0
2010-01-07 NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the `fillna()` method.

See the user guide for more.
DataFrame.reindex_like

DataFrame.reindex_like(other, method=None, copy=True, limit=None, tolerance=None)

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other [Object of the same data type] Its row and column indices are used to define the new indices of this object.

method [(None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’)] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

• None (default): don’t fill gaps
• pad / ffill: propagate last valid observation forward to next valid
• backfill / bfill: use next valid observation to fill gap
• nearest: use nearest valid observations to fill gap.

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation \( \text{abs(index[indexer] - target)} \leq \text{tolerance} \).

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

Returns

Series or DataFrame Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex Change to new indices or expand indices.
Notes

Same as calling .reindex(index=other.index, columns=other.columns,...).

Examples

```python
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
                      ... [31, 87.8, 'high'],
                      ... [22, 71.6, 'medium'],
                      ... [35, 95, 'medium']],
                     ... columns=['temp_celsius', 'temp_fahrenheit',
                     ... 'windspeed'],
                     ... index=pd.date_range(start='2014-02-12',
                     ... end='2014-02-15', freq='D'))
```

```
>>> df1
                      temp_celsius  temp_fahrenheit  windspeed
2014-02-12            24.3            75.7  high
2014-02-13            31.0            87.8  high
2014-02-14            22.0            71.6  medium
2014-02-15            35.0            95.0  medium
```

```python
>>> df2 = pd.DataFrame([[28, 'low'],
                      ... [30, 'low'],
                      ... [35.1, 'medium']],
                     ... columns=['temp_celsius', 'windspeed'],
                     ... index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
                     ... '2014-02-15']))
```

```
>>> df2
                      temp_celsius  windspeed
2014-02-12            28.0  low
2014-02-13            30.0  low
2014-02-15            35.1  medium
```

```python
>>> df2.reindex_like(df1)
                      temp_celsius  temp_fahrenheit  windspeed
2014-02-12            28.0            NaN  low
2014-02-13            30.0            NaN  low
2014-02-14            NaN            NaN  NaN
2014-02-15            35.1            NaN  medium
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.DataFrame.rename

DataFrame.rename(mapper=None, *, index=None, columns=None, axis=None, copy=True, inplace=False, level=None, errors='ignore')

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

Parameters

mapper [dict-like or function] Dict-like or function transformations to apply to that axis' values. Use either mapper and axis to specify the axis to target with mapper, or index and columns.

index [dict-like or function] Alternative to specifying axis (mapper, axis=0 is equivalent to index=mapper).

columns [dict-like or function] Alternative to specifying axis (mapper, axis=1 is equivalent to columns=mapper).

axis [[0 or ‘index’, 1 or ‘columns’], default 0] Axis to target with mapper. Can be either the axis name (‘index’, ‘columns’) or number (0, 1). The default is ‘index’.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

errors [['ignore', 'raise'], default ‘ignore’] If ‘raise’, raise a KeyError when a dict-like mapper, index, or columns contains labels that are not present in the Index being transformed. If ‘ignore’, existing keys will be renamed and extra keys will be ignored.

Returns

DataFrame or None DataFrame with the renamed axis labels or None if inplace=True.

Raises

KeyError If any of the labels is not found in the selected axis and “errors=’raise’”.

See also:

DataFrame.rename_axis Set the name of the axis.
Examples

DataFrame.rename supports two calling conventions

- (index=index_mapper, columns=columns_mapper, ...)
- (mapper, axis={'index', 'columns'}, ...)

We highly recommend using keyword arguments to clarify your intent.

Rename columns using a mapping:

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
     a  c
0  1  4
1  2  5
2  3  6
```

Rename index using a mapping:

```python
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
     A  B
   x  1  4
   y  2  5
   z  3  6
```

Cast index labels to a different type:

```python
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index([0, 1, 2], dtype='object')
```

```python
>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
  File "", line 19, in <module>
KeyError: ['C'] not found in axis
```

Using axis-style parameters:

```python
>>> df.rename(str.lower, axis='columns')
     a  b
0  1  4
1  2  5
2  3  6
```

```python
>>> df.rename({1: 2, 2: 4}, axis='index')
     A  B
0  1  4
2  2  5
4  3  6
```
DataFrame.rename_axis(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False)

Set the name of the axis for the index or columns.

Parameters:

mapper [scalar, list-like, optional] Value to set the axis name attribute.

index, columns [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the columns parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either mapper and axis to specify the axis to target with mapper, or index and/or columns.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to rename.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns:

Series, DataFrame, or None The same type as the caller or None if inplace=True.

See also:

Series.rename Alter Series index labels or name.

DataFrame.rename Alter DataFrame index labels or name.

Index.rename Set new names on index.

Notes:

DataFrame.rename_axis supports two calling conventions

• (index=index_mapper, columns=columns_mapper, ...)
• (mapper, axis={'index', 'columns'}, ...)

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter copy is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis labels.

We highly recommend using keyword arguments to clarify your intent.
Examples

Series

```python
>>> s = pd.Series(['dog', 'cat', 'monkey'])
>>> s
0  dog
1  cat
2  monkey
dtype: object
>>> s.rename_axis('animal')
animal
0  dog
1  cat
2  monkey
dtype: object
```

Dataframe

```python
>>> df = pd.DataFrame({
    'num_legs': [4, 4, 2],
    'num_arms': [0, 0, 2]
},
... [
    'dog', 'cat', 'monkey']
)
>>> df
num_legs  num_arms
dog       4  0
cat       4  0
monkey    2  2
```

```python
>>> df = df.rename_axis('animal')
>>> df
num_legs  num_arms
animal
dog       4  0
cat       4  0
monkey    2  2
```

```python
>>> df = df.rename_axis('limbs', axis='columns')
>>> df
limbs  num_legs  num_arms
animal
dog    4  0
cat    4  0
monkey 2  2
```

MultiIndex

```python
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...     ['dog', 'cat', 'monkey']],
...     names=['type', 'name'])
>>> df
limbs  num_legs  num_arms
type  name
mammal dog    4  0
        cat    4  0
        monkey 2  2
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.rename_axis(index={'type': 'class'})
limbs  num_legs  num_arms
class  name
mammal dog     4     0
    cat     4     0
   monkey   2     2
```

```python
>>> df.rename_axis(columns=str.upper)
LIMBS  num_legs  num_arms
type  name
mammal dog     4     0
    cat     4     0
   monkey   2     2
```

### pandas.DataFrame.reorder_levels

#### DataFrame.reorder_levels(order, axis=0)

Rearrange index levels using input order. May not drop or duplicate levels.

**Parameters**

- **order** [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).
- **axis** [[0 or ‘index’, 1 or ‘columns’], default 0] Where to reorder levels.

**Returns**

DataFrame

**Examples**

```python
>>> data = {
>     "class": ["Mammals", "Mammals", "Reptiles"],
>     "diet": ["Omnivore", "Carnivore", "Carnivore"],
>     "species": ["Humans", "Dogs", "Snakes"],
> }
>> df = pd.DataFrame(data, columns=["class", "diet", "species"])
>> df = df.set_index(["class", "diet"])
>> df
     species       class  diet
Mammals  Humans  Omnivore
       Dogs  Carnivore
Reptiles  Snakes  Carnivore
```

Let's reorder the levels of the index:

```python
>>> df.reorder_levels(["diet", "class"])
     species       class  diet
Omnivore  Humans  Mammals
```

(continues on next page)
pandas.DataFrame.replace

DataFrame.replace(to_replace=None, value=NoneDefault.no_default, inplace=False, limit=None, regex=False, method=NoneDefault.no_default)

Replace values given in to_replace with value.

Values of the DataFrame are replaced with other values dynamically.

This differs from updating with .loc or .iloc, which require you to specify a location to update with some value.

Parameters

- to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.
  - numeric, str or regex:
    - numeric: numeric values equal to to_replace will be replaced with value
    - str: string exactly matching to_replace will be replaced with value
    - regex: regexs matching to_replace will be replaced with value
  - list of str, regex, or numeric:
    - First, if to_replace and value are both lists, they must be the same length.
    - Second, if regex=True then all of the strings in both lists will be interpreted as regexes otherwise they will match directly. This doesn’t matter much for value since there are only a few possible substitution regexes you can use.
    - str, regex and numeric rules apply as above.
  - dict:
    - Dicts can be used to specify different replacement values for different existing values. For example, {'a': 'b', 'y': 'z'} replaces the value ‘a’ with ‘b’ and ‘y’ with ‘z’. To use a dict in this way the value parameter should be None.
    - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, {'a': 1, 'b': 'z'} looks for the value 1 in column ‘a’ and the value ‘z’ in column ‘b’ and replaces these values with whatever is specified in value. The value parameter should not be None in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
    - For a DataFrame nested dictionaries, e.g., {'a': {'b': np.nan}}, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with NaN. The value parameter should be None to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.
  - None:
This means that the regex argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If value is also None then this must be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** [scalar, dict, list, str, regex, default None] Value to replace any values matching to_replace with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dict of such objects are also allowed.

**inplace** [bool, default False] If True, performs operation inplace and returns None.

**limit** [int, default None] Maximum size gap to forward or backward fill.

**regex** [bool or same types as to_replace, default False] Whether to interpret to_replace and/or value as regular expressions. If this is True then to_replace must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case to_replace must be None.

**method** [{‘pad’, ‘ffill’, ‘bfill’, None}] The method to use when for replacement, when to_replace is a scalar, list or tuple and value is None.

Changed in version 0.23.0: Added to DataFrame.

**Returns**

DataFrame Object after replacement.

**Raises**

AssertionError

- If regex is not a bool and to_replace is not None.

TypeError

- If to_replace is not a scalar, array-like, dict, or None
- If to_replace is a dict and value is not a list, dict, ndarray, or Series
- If to_replace is None and regex is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple bool or datetime64 objects and the arguments to to_replace does not match the type of the value being replaced

ValueError

- If a list or an ndarray is passed to to_replace and value but they are not the same length.

**See also:**

DataFrame.fillna Fill NA values.

DataFrame.where Replace values based on boolean condition.

Series.str.replace Simple string replacement.
Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers are strings, then you can do this.
- This method has a lot of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the `to_replace` value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar `to_replace` and `value`:

```python
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```python
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
                      'B': [5, 6, 7, 8, 9],
                      'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
     A  B  C
0    5  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

List-like `to_replace`:

```python
>>> df.replace([0, 1, 2, 3], 4)
     A  B  C
0    4  5  a
1    4  6  b
2    4  7  c
3    4  8  d
4    4  9  e
```

```python
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
     A  B  C
0    4  5  a
1    4  6  b
2    4  7  c
3    4  8  d
4    4  9  e
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

>>> s.replace([1, 2], method='bfill')
0  3
1  3
2  3
3  4
4  5
dtype: int64

```
dict-like 'to_replace'
```

>>> df.replace({0: 10, 1: 100})
   A  B  C
0  10  5  a
1 100  6  b
2  2  7  c
3  3  8  d
4  4  9  e

>>> df.replace({'A': 0, 'B': 5}, 100)
   A  B  C
0  100  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e

>>> df.replace({'A': {'0': 100, '4': 400}})
   A  B  C
0 100  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4 400  9  e

```
Regular expression 'to_replace'
```

>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})

```python
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
   A  B
0 new abc
1 foo new
2  bait xyz
```

>>> df.replace({'A': r'^ba.$', 'A': 'new'}, regex=True)
   A  B
0 new abc

(continues on next page)
1 foo bar
2 bait xyz

```python
>>> df.replace(regex=r'^ba.$', value='new')
   A   B
0  new  abc
1    foo  new
2   bait  xyz
```

```python
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
   A   B
0  new  abc
1  xyz  new
2   bait  xyz
```

```python
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
   A   B
0  new  abc
1   new  new
2   bait  xyz
```

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```python
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the `value` parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None):

```python
>>> s.replace({'a': None})
0   10
1  None
2  None
3   b
4  None
dtype: object
```

When `value` is not explicitly passed and `to_replace` is a scalar, list or tuple, `replace` uses the method parameter (default 'pad') to do the replacement. So this is why the ‘a’ values are being replaced by 10 in rows 1 and 2 and ‘b’ in row 4 in this case.

```python
>>> s.replace('a')
0   10
1   10
2   10
3   b
4   b
dtype: object
```

On the other hand, if `None` is explicitly passed for `value`, it will be respected:
```python
>>> s.replace('a', None)
0    10
1    None
2    None
3     b
4    None
dtype: object
```

Changed in version 1.4.0: Previously the explicit `None` was silently ignored.

**pandas.DataFrame.resample**

```python
dataFrame.resample(rule, axis=0, closed=None, label=None, convention='start', kind=None, 
loffset=None, base=None, on=None, level=None, origin='start_day', offset=None)
```

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a
datetime-like index (`DatetimeIndex`, `PeriodIndex`, or `TimedeltaIndex`), or the caller must pass the label of
a datetime-like series/index to the `on/level` keyword parameter.

**Parameters**

- `rule` [DateOffset, Timedelta or str] The offset string or object representing target conversion.
- `axis` [{0 or ‘index’, 1 or ‘columns’}, default 0] Which axis to use for up- or down-
sampling. For Series this will default to 0, i.e. along the rows. Must be `DatetimeIn-
dex`, `TimedeltaIndex` or `PeriodIndex`.
- `closed` [{‘right’, ‘left’}, default None] Which side of bin interval is closed. The default
  which all have a default of ‘right’.
- `label` [{‘right’, ‘left’}, default None] Which bin edge label to label bucket with. The
  and ‘W’ which all have a default of ‘right’.
- `convention` [{‘start’, ‘end’, ‘s’, ‘e’}, default ‘start’] For `PeriodIndex` only, controls
  whether to use the start or end of `rule`.
- `kind` [{‘timestamp’, ‘period’}, optional, default None] Pass ‘timestamp’ to convert the
  resulting index to a `DateTimeIndex` or ‘period’ to convert it to a `PeriodIndex`. By
default the input representation is retained.
- `loffset` [timedelta, default None] Adjust the resampled time labels.
  Deprecated since version 1.1.0: You should add the loffset to the `df.index` after the
  resample. See below.
- `base` [int, default 0] For frequencies that evenly subdivide 1 day, the “origin” of the
  aggregated intervals. For example, for ‘5min’ frequency, base could range from 0
  through 4. Defaults to 0.
  Deprecated since version 1.1.0: The new arguments that you should use are ‘offset’
  or ‘origin’.
- `on` [str, optional] For a DataFrame, column to use instead of index for resampling. Col-
must be datetime-like.
**level** [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. `level` must be datetime-like.

**origin** [Timestamp or str, default ‘start_day’] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:
- ‘epoch’: `origin` is 1970-01-01
- ‘start’: `origin` is the first value of the timeseries
- ‘start_day’: `origin` is the first day at midnight of the timeseries

New in version 1.1.0.
- ‘end’: `origin` is the last value of the timeseries
- ‘end_day’: `origin` is the ceiling midnight of the last day

New in version 1.3.0.

**offset** [Timedelta or str, default is None] An offset timedelta added to the origin.

New in version 1.1.0.

**Returns**

`pandas.core.Resampler` Resampler object.

**See also:**

`Series.resample` Resample a Series.

`DataFrame.resample` Resample a DataFrame.

`groupby` Group DataFrame by mapping, function, label, or list of labels.

`asfreq` Reindex a DataFrame with the given frequency without grouping.

**Notes**

See the user guide for more.

To learn more about the offset strings, please see this link.

**Examples**

Start by creating a series with 9 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
```

(continues on next page)
Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```python
>>> series.resample('3T').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket 2000-01-01 00:03:00 contains the value 3, but the summed value in the resampled bucket with the label 2000-01-01 00:03:00 does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```python
>>> series.resample('3T', label='right').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```python
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```python
>>> series.resample('30S').asfreq()[0:5]  # Select first 5 rows
2000-01-01 00:00:00    0
2000-01-01 00:03:30  NaN
2000-01-01 00:01:00    1
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00    2
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the NaN values using the pad method.

```python
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the NaN values using the bfill method.
Pass a custom function via apply

```python
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
... >>> series.resample('3T').apply(custom_resampler)
```

```
2000-01-01 00:00:00 8
2000-01-01 00:03:00 17
2000-01-01 00:06:00 26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using *start* *convention*. Values are assigned to the first quarter of the period.

```python
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                 freq='A',
...                 periods=2))
>>> s
2012 1
2013 2
Freq: A-DEC, dtype: int64
```

```python
>>> s.resample('Q', convention='start').asfreq()
```

```
2012Q1 1.0
2012Q2 NaN
2012Q3 NaN
2012Q4 NaN
2013Q1 2.0
2013Q2 NaN
2013Q3 NaN
2013Q4 NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using *end* *convention*. Values are assigned to the last month of the period.

```python
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                 freq='Q',
...                 periods=4))
>>> q
2018Q1 1
2018Q2 2
2018Q3 3
2018Q4 4
Freq: Q-DEC, dtype: int64
```

```python
>>> q.resample('M', convention='end').asfreq()
```
For DataFrame objects, the keyword \texttt{on} can be used to specify the column instead of the index for resampling.

\begin{verbatim}
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018', periods=8, freq='W')
>>> df

price  volume  week_starting
0     10     50  2018-01-07
1     11     60  2018-01-14
2      9     40  2018-01-21
3     13    100  2018-01-28
4     14     50  2018-02-04
5     18    100  2018-02-11
6     17     40  2018-02-18
7     19     50  2018-02-25

>>> df.resample('M', on='week_starting').mean()

price  volume
week_starting
2018-01-31  10.75  62.5
2018-02-28  17.00  60.0
\end{verbatim}

For a DataFrame with MultiIndex, the keyword \texttt{level} can be used to specify on which level the resampling needs to take place.

\begin{verbatim}
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(d2,
                    index=pd.MultiIndex.from_product([days, ['morning', 'afternoon']]))

price  volume
2000-01-01 morning  10  50
\end{verbatim}
afternoon 11 60
2000-01-02 morning 9 40
afternoon 13 100
2000-01-03 morning 14 50
afternoon 18 100
2000-01-04 morning 17 40
afternoon 19 50

```python
>>> df2.resample('D', level=0).sum()
    price  volume
2000-01-01 21  110
2000-01-02 22  140
2000-01-03 32  150
2000-01-04 36   90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```python
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00    0
2000-10-01 23:37:00    3
2000-10-01 23:44:00    6
2000-10-01 23:51:00    9
2000-10-01 23:58:00   12
2000-10-02 00:05:00   15
2000-10-02 00:12:00   18
2000-10-02 00:19:00   21
2000-10-02 00:26:00   24
Freq: 7T, dtype: int64

```python
>>> ts.resample('17min').sum()
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
2000-10-02 00:22:00   24
Freq: 17T, dtype: int64
```

```python
>>> ts.resample('17min', origin='epoch').sum()
2000-10-01 23:18:00    0
2000-10-01 23:35:00   18
2000-10-01 23:52:00   27
2000-10-02 00:09:00   39
2000-10-02 00:26:00   24
Freq: 17T, dtype: int64
```

```python
>>> ts.resample('17min', origin='2000-01-01').sum()
2000-10-01 23:24:00    3
2000-10-01 23:41:00   15
2000-10-01 23:58:00   45
```

(continues on next page)
2000-10-02 00:15:00 45
Freq: 17T, dtype: int64

If you want to adjust the start of the bins with an offset Timedelta, the two following lines are equivalent:

```python
>>> ts.resample('17min', origin='start').sum()
2000-10-01 23:30:00 9
2000-10-01 23:47:00 21
2000-10-02 00:04:00 54
2000-10-02 00:21:00 24
Freq: 17T, dtype: int64
```

```python
>>> ts.resample('17min', offset='23h30min').sum()
2000-10-01 23:30:00 9
2000-10-01 23:47:00 21
2000-10-02 00:04:00 54
2000-10-02 00:21:00 24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```python
>>> ts.resample('17min', origin='end').sum()
2000-10-01 23:35:00 0
2000-10-01 23:52:00 18
2000-10-02 00:09:00 27
2000-10-02 00:26:00 63
Freq: 17T, dtype: int64
```

In contrast with the start_day, you can use end_day to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```python
>>> ts.resample('17min', origin='end_day').sum()
2000-10-01 23:38:00 3
2000-10-01 23:55:00 15
2000-10-02 00:12:00 45
2000-10-02 00:29:00 45
Freq: 17T, dtype: int64
```

To replace the use of the deprecated base argument, you can now use offset, in this example it is equivalent to have base=2:

```python
>>> ts.resample('17min', offset='2min').sum()
2000-10-01 23:16:00 0
2000-10-01 23:33:00 9
2000-10-01 23:50:00 36
2000-10-02 00:07:00 39
2000-10-02 00:24:00 24
Freq: 17T, dtype: int64
```

To replace the use of the deprecated loffset argument:

```python
>>> from pandas.tseries.frequencies import to_offset
>>> loffset = '19min'
```
>>> ts_out = ts.resample('17min').sum()
>>> ts_out.index = ts_out.index + to_offset(loffset)
>>> ts_out
   2000-10-01 23:33:00  0
   2000-10-01 23:50:00  9
   2000-10-02 00:07:00 21
   2000-10-02 00:24:00 54
   2000-10-02 00:41:00 24
Freq: 17T, dtype: int64

**pandas.DataFrame.reset_index**

`DataFrame.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill=None)`

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

**Parameters**

- **level** [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.
- **drop** [bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.
- **inplace** [bool, default False] Modify the DataFrame in place (do not create a new object).
- **col_level** [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.
- **col_fill** [object, default ‘’] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

**Returns**

- **DataFrame or None** DataFrame with the new index or None if `inplace=True`.

**See also:**

- `DataFrame.set_index` Opposite of reset_index.
- `DataFrame.reindex` Change to new indices or expand indices.
- `DataFrame.reindex_like` Change to same indices as other DataFrame.
Examples

```python
>>> df = pd.DataFrame([
    ('bird', 389.0),
    ('bird', 24.0),
    ('mammal', 80.5),
    ('mammal', np.nan),
],
    index=['falcon', 'parrot', 'lion', 'monkey'],
    columns=('class', 'max_speed'))
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```python
>>> df.reset_index()
```

<table>
<thead>
<tr>
<th>index</th>
<th>class</th>
<th>max_speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>falcon</td>
<td>389.0</td>
</tr>
<tr>
<td>1</td>
<td>parrot</td>
<td>24.0</td>
</tr>
<tr>
<td>2</td>
<td>lion</td>
<td>80.5</td>
</tr>
<tr>
<td>3</td>
<td>monkey</td>
<td>NaN</td>
</tr>
</tbody>
</table>

We can use the `drop` parameter to avoid the old index being added as a column:

```python
>>> df.reset_index(drop=True)
```

<table>
<thead>
<tr>
<th>class</th>
<th>max_speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td>389.0</td>
</tr>
<tr>
<td>bird</td>
<td>24.0</td>
</tr>
<tr>
<td>mammal</td>
<td>80.5</td>
</tr>
<tr>
<td>mammal</td>
<td>NaN</td>
</tr>
</tbody>
</table>

You can also use `reset_index` with `MultiIndex`.

```python
>>> index = pd.MultiIndex.from_tuples(
    [(bird, 'falcon'),
     (bird, 'parrot'),
     (mammal, 'lion'),
     (mammal, 'monkey')],
    names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples(
    [('speed', 'max'),
     ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
     (24.0, 'fly'),
     (80.5, 'run'),
     (np.nan, 'jump')],
    index=index,
    columns=columns)
```

<table>
<thead>
<tr>
<th>speed</th>
<th>species</th>
</tr>
</thead>
<tbody>
<tr>
<td>389.0</td>
<td>fly</td>
</tr>
<tr>
<td>24.0</td>
<td>fly</td>
</tr>
<tr>
<td>80.5</td>
<td>run</td>
</tr>
<tr>
<td>NaN</td>
<td>jump</td>
</tr>
</tbody>
</table>

(continues on next page)
If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
name    class  speed  species
        max    type
falcon  bird  389.0  fly
parrot  bird  24.0  fly
lion    mammal  80.5  run
monkey  mammal  NaN  jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
name    class  max    type
falcon  bird  389.0  fly
parrot  bird  24.0  fly
lion    mammal  80.5  run
monkey  mammal  NaN  jump
```

When the index is inserted under another level, we can specify under which one with the parameter `col_fill`:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
name    species  speed  species
        class  max    type
falcon  bird    389.0  fly
parrot  bird    24.0  fly
lion    mammal  80.5  run
monkey  mammal  NaN  jump
```

If we specify a nonexistent level for `col_fill`, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
name    genus  speed  species
        class  max    type
falcon  bird    389.0  fly
parrot  bird    24.0  fly
lion    mammal  80.5  run
monkey  mammal  NaN  jump
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DataFrame.rfloordiv**

`DataFrame.rfloordiv(other, axis='columns', level=None, fill_value=None)`

Get integer division of dataframe and other, element-wise (binary operator `rfloordiv`).

Equivalent to `other // dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs. With reverse version, `floordiv`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or ‘index’, 1 or ‘columns’}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

`DataFrame` Result of the arithmetic operation.

**See also:**

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.
Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
                      angles  degrees
    circle            0      360
    triangle          3      180
    rectangle         4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
                      angles  degrees
    circle            1      361
    triangle          4      181
    rectangle         5      361
```

```python
>>> df.add(1)
                      angles  degrees
    circle            1      361
    triangle          4      181
    rectangle         5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
                      angles  degrees
    circle          0.0      36.0
    triangle        0.3      18.0
    rectangle       0.4      36.0
```

```python
>>> df.rdiv(10)
                      angles  degrees
    circle        inf      0.027778
    triangle    3.333333    0.055556
    rectangle  2.500000    0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
                      angles  degrees
    circle           -1      358
    triangle          2      178
    rectangle         3      358
```

```python
>>> df.sub([1, 2], axis='columns')
                      angles  degrees
    circle           -1      358
    triangle          2      178
    rectangle         3      358
```

3.4. DataFrame

1875
## Multiplying a DataFrame of Different Shape

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({
    'angles': [0, 3, 4],
    'degrees': [360, 180, 360, 360, 540, 720],
}, index=['A', 'A', 'B', 'B', 'B'],
                    ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
```

```python
>>> df * other
angles | degrees
--- | ---
circle | 0 | NaN
triangle | 9 | NaN
rectangle | 16 | NaN
```

```python
>>> df.mul(other, fill_value=0)
angles | degrees
--- | ---
circle | 0 | 0.0
triangle | 9 | 0.0
rectangle | 16 | 0.0
```

## Dividing by a MultiIndex by Level

Divide by a MultiIndex by level.

```python
def_multindex = pd.DataFrame({
    'angles': [0, 3, 4, 5, 6],
    'degrees': [360, 180, 360, 360, 540, 720],
}, index=['A', 'A', 'B', 'B', 'B'],
                    ['circle', 'triangle', 'rectangle',
                     'square', 'pentagon', 'hexagon'])
```

```python
>>> df_multindex
angles | degrees
--- | ---
A circle | 0 | 360
triangle | 3 | 180
rectangle | 4 | 360
B square | 4 | 360
pentagon | 5 | 540
hexagon | 6 | 720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
angles | degrees
--- | ---
A circle | NaN | 1.0
triangle | 1.0 | 1.0
rectangle | 1.0 | 1.0
B square | 0.0 | 0.0
pentagon | 0.0 | 0.0
hexagon | 0.0 | 0.0
```
DataFrame.rmod

DataFrame.rmod(other, axis='columns', level=None, fill_value=None)

Get Modulo of dataframe and other, element-wise (binary operator rmod).

Equivalent to other % dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, mod.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -., /*, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.
Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
                  angles  degrees
    circle        0       360
    triangle      3       180
    rectangle     4       360

Add a scalar with operator version which return the same results.

```python
>>> df + 1
                  angles  degrees
    circle        1       361
    triangle      4       181
    rectangle     5       361
```python

```python
>>> df.add(1)  
                  angles  degrees
    circle        1       361
    triangle      4       181
    rectangle     5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
                  angles  degrees
    circle      0.0      36.0
    triangle    0.3      18.0
    rectangle   0.4      36.0
```python

```python
>>> df.rdiv(10)
                  angles  degrees
    circle      inf      0.027778
    triangle    3.333333  0.055556
    rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
                  angles  degrees
    circle       -1       358
    triangle      2       178
    rectangle     3       358
```python

```python
>>> df.sub([1, 2], axis='columns')
                  angles  degrees
    circle       -1       358
    triangle      2       178
    rectangle     3       358
```
Multiply a DataFrame of different shape with operator version.

```python
df * other
```

```plaintext
angles  degrees
circle  0   NaN
triangle 9   NaN
rectangle 16  NaN
```

```python
df.mul(other, fill_value=0)
```

```plaintext
angles  degrees
0       0.0
9       0.0
16      0.0
```

Divide by a MultiIndex by level.

```python
df.div(df_multindex, level=1, fill_value=0)
```

```plaintext
angles  degrees
A circle 0 1.0
triangle 1.0 1.0
rectangle 1.0 1.0
B square 0.0 0.0
pentagon 0.0 0.0
hexagon 0.0 0.0
```

3.4. DataFrame
pandasDataFrame.rmul

**DataFrame.rmul**(other, axis='columns', level=None, fill_value=None)

Get Multiplication of dataframe and other, element-wise (binary operator rmul).

Equivalent to other * dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, mul.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.

**See also:**

- **DataFrame.add** Add DataFrames.
- **DataFrame.sub** Subtract DataFrames.
- **DataFrame.mul** Multiply DataFrames.
- **DataFrame.div** Divide DataFrames (float division).
- **DataFrame.truediv** Divide DataFrames (float division).
- **DataFrame.floordiv** Divide DataFrames (integer division).
- **DataFrame.mod** Calculate modulo (remainder after division).
- **DataFrame.pow** Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.
Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
          angles  degrees
    circle     0       360
     triangle   3       180
    rectangle   4       360
Add a scalar with operator version which return the same results.

>>> df + 1
          angles  degrees
    circle     1       361
     triangle   4       181
    rectangle   5       361

>>> df.add(1)
          angles  degrees
    circle     1       361
     triangle   4       181
    rectangle   5       361
Divide by constant with reverse version.

>>> df.div(10)
          angles  degrees
    circle   0.0       36.0
     triangle 0.3       18.0
    rectangle 0.4       36.0

>>> df.rdiv(10)
          angles  degrees
    circle   inf       0.027778
     triangle 3.33333  0.055556
    rectangle 2.50000  0.027778
Subtract a list and Series by axis with operator version.

>>> df - [1, 2]
          angles  degrees
    circle   -1       358
     triangle    2       178
    rectangle    3       358

>>> df.sub([1, 2], axis='columns')
          angles  degrees
    circle   -1       358
     triangle    2       178
    rectangle    3       358
```
Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4],
                        'degrees': [360, 180, 360, 360, 540, 720],
                        'index': ['A', 'A', 'A', 'B', 'B', 'B'],
                        'level': ['circle', 'triangle', 'rectangle',
                                  'square', 'pentagon', 'hexagon']})
>>> df_multindex
         angles degrees
A circle  0   360
triangle  3   180
rectangle 4   360
B square  4   360
pentagon  5   540
hexagon  6   720
```

Divide by a MultiIndex by level.

```python
>>> df_multindex
         angles degrees
A circle  0   360
triangle  3   180
rectangle 4   360
B square  4   360
pentagon  5   540
hexagon  6   720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
         angles degrees
A circle  NaN   1.0
triangle  1.0   1.0
rectangle 1.0   1.0
B square  0.0   0.0
pentagon  0.0   0.0
hexagon  0.0   0.0
```
DataFrame.rolling

DataFrame.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0, closed=None, method='single')

Provide rolling window calculations.

**Parameters**

- **window** [int, offset, or BaseIndexer subclass] Size of the moving window.
  - If an integer, the fixed number of observations used for each window.
  - If an offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. To learn more about the offsets & frequency strings, please see this link.
  - If a BaseIndexer subclass, the window boundaries based on the defined get_window_bounds method. Additional rolling keyword arguments, namely min_periods, center, and closed will be passed to get_window_bounds.

- **min_periods** [int, default None] Minimum number of observations in window required to have a value; otherwise, result is np.nan.
  - For a window that is specified by an offset, min_periods will default to 1.
  - For a window that is specified by an integer, min_periods will default to the size of the window.

- **center** [bool, default False] If False, set the window labels as the right edge of the window index.
  - If True, set the window labels as the center of the window index.

- **win_type** [str, default None] If None, all points are evenly weighted.
  - If a string, it must be a valid scipy.signal window function.
  - Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature.

- **on** [str, optional] For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame’s index.
  - Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

- **axis** [int or str, default 0] If 0 or 'index', roll across the rows.
  - If 1 or 'columns', roll across the columns.

- **closed** [str, default None] If 'right', the first point in the window is excluded from calculations.
  - If 'left', the last point in the window is excluded from calculations.
  - If 'both', the no points in the window are excluded from calculations.
  - If 'neither', the first and last points in the window are excluded from calculations.
  - Default None ('right').

Changed in version 1.2.0: The closed parameter with fixed windows is now supported.
**method** [str {'single', 'table'}, default 'single'] New in version 1.3.0.

Execute the rolling operation per single column or row ('single') or over the entire object ('table').

This argument is only implemented when specifying engine='numba' in the method call.

Returns

*Window subclass if a win_type is passed*

*Rolling subclass if win_type is not passed*

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions.

**Notes**

See Windowing Operations for further usage details and examples.

**Examples**

```python
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3 NaN
4  4.0

window

Rolling sum with a window length of 2 observations.

```python
def.rolling(2).sum()

```
   B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN

Rolling sum with a window span of 2 seconds.

```python
def_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
                        index=[pd.Timestamp('20130101 09:00:00'),
                               pd.Timestamp('20130101 09:00:02'),
                               pd.Timestamp('20130101 09:00:03'),
                               pd.Timestamp('20130101 09:00:05'),
                               pd.Timestamp('20130101 09:00:06')])
```
>>> df_time
  B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

>>> df_time.rolling('2s').sum()
  B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

Rolling sum with forward looking windows with 2 observations.

>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
   B
0   1.0
1   3.0
2   2.0
3   4.0
4   4.0

min_periods
Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

>>> df.rolling(2, min_periods=1).sum()
   B
0   0.0
1   1.0
2   3.0
3   2.0
4   4.0

center
Rolling sum with the result assigned to the center of the window index.

>>> df.rolling(3, min_periods=1, center=True).sum()
   B
0   1.0
1   3.0
2   3.0
3   6.0
4   4.0

>>> df.rolling(3, min_periods=1, center=False).sum()
   B
(continues on next page)
win_type

Rolling sum with a window length of 2, using the Scipy 'gaussian' window type. std is required in the aggregation function.

```python
>>> df.rolling(2, win_type='gaussian').sum(std=3)
   B
0  NaN
1  0.986207
2  2.958621
3  NaN
4  NaN
```

**pandas.DataFrame.round**

Dataframe.round(\(decimals=0, *\text{args}, **\text{kwargs}\))

Round a DataFrame to a variable number of decimal places.

**Parameters**

- **decimals** [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if \(decimals\) is a dict-like, or in the index if \(decimals\) is a Series. Any columns not included in \(decimals\) will be left as is. Elements of \(decimals\) which are not columns of the input will be ignored.

- *\text{args} Additional keywords have no effect but might be accepted for compatibility with numpy.

- **\text{kwargs} Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns**

- **Dataframe** A DataFrame with the affected columns rounded to the specified number of decimal places.

**See also:**

- **numpy.\text{round}** Round a numpy array to the given number of decimals.

- **Series.\text{round}** Round a Series to the given number of decimals.
Examples

```python
>>> df = pd.DataFrame((.21, .32), (.01, .67), (.66, .03), (.21, .18)),
...     columns=['dogs', 'cats'])
```

```text
By providing an integer each column is rounded to the same number of decimal places
```

```python
>>> df.round(1)
```

```text
With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value
```

```python
>>> df.round({'dogs': 1, 'cats': 0})
```

```text
Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value
```

```python
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
```

```
pandas.DataFrame.rpow
```

DataFrame.rpow(other, axis='columns', level=None, fill_value=None)

Get Exponential power of dataframe and other, element-wise (binary operator rpow).

Equivalent to other ** dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, pow.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis ([0 or `index`, 1 or `columns`]) Whether to compare by the index (0 or `index`’) or columns (1 or `columns`). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

- **DataFrame** Result of the arithmetic operation.

See also:

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'])
>>> df
angles  degrees
circle   0      360
triangle 3      180
rectangle 4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
angles  degrees
circle   1      361
triangle 4      181
rectangle 5      361
```
>>> df.add(1)
    angles  degrees
   circle     1      361
          4      181
   rectangle    5      361

Divide by constant with reverse version.

>>> df.div(10)
    angles  degrees
   circle     0.0      36.0
          0.3      18.0
   rectangle    0.4      36.0

>>> df.rdiv(10)
    angles  degrees
   circle    inf      0.027778
          3.333333      0.055556
   rectangle  2.500000      0.027778

Subtract a list and Series by axis with operator version.

>>> df - [1, 2]
    angles  degrees
   circle    -1      358
          2      178
   rectangle    3      358

>>> df.sub([1, 2], axis='columns')
    angles  degrees
   circle    -1      358
          2      178
   rectangle    3      358

>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
        axis='index')
    angles  degrees
   circle    -1      359
          2      179
   rectangle    3      359

Multiply a DataFrame of different shape with operator version.

>>> other = pd.DataFrame({'angles': [0, 3, 4],
                         'index=['circle', 'triangle', 'rectangle']})
>>> other
    angles
   circle    0
   triangle    3
   rectangle    4
pandas DataFrame rsub

DataFrame.rsub(other, axis='columns', level=None, fill_value=None)

Get Subtraction of dataframe and other, element-wise (binary operator rsub).

Equivalent to other - dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, sub.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns' }] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'])
```

```
angles  degrees
circle   0   360
triangle 3   180
rectangle 4   360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
```

```
angles  degrees
circle   1   361
triangle 4   181
rectangle 5   361
```

```
>>> df.add(1)
```

```
angles  degrees
circle   1   361
```

(continues on next page)
Divide by constant with reverse version.

```python
>>> df.div(10)
angles degrees
circle 0.0 36.0
triangle 0.3 18.0
rectangle 0.4 36.0
```

```python
>>> df.rdiv(10)
angles degrees
circle inf 0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
angles degrees
circle -1 358
triangle 2 178
rectangle 3 358
```

```python
>>> df.sub([1, 2], axis='columns')
angles degrees
circle -1 358
triangle 2 178
rectangle 3 358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
axis='index')
angles degrees
circle -1 359
triangle 2 179
rectangle 3 359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({
'angles': [0, 3, 4],
index=['circle', 'triangle', 'rectangle']})
```

```python
>>> df * other
angles degrees
circle 0 NaN
```
triangle 9 NaN
rectangle 16 NaN

```python
>>> df.mul(other, fill_value=0)
angles  degrees
circle  0  0.0
triangle 9  0.0
rectangle 16 0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                               'degrees': [360, 180, 360, 540, 720],
                               'index':[['A', 'A', 'A', 'B', 'B'],
                                        ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon']])
```

```python
>>> df_div = df_multindex
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
angles  degrees
A cirle  NaN  1.0
triangle 1.0  1.0
rectangle 1.0  1.0
B square 0.0  0.0
pentagon 0.0  0.0
hexagon 0.0  0.0
```

**pandas.DataFrame.rtruediv**

**DataFrame.rtruediv**(other, axis=’columns’, level=None, fill_value=None)  
Get Floating division of dataframe and other, element-wise (binary operator `rtruediv`).  
Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, `truediv`.  
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [0 or ‘index’, 1 or ‘columns’]) Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
**fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.

**See also:**

*DataFrame.add* Add DataFrames.
*DataFrame.sub* Subtract DataFrames.
*DataFrame.mul* Multiply DataFrames.
*DataFrame.div* Divide DataFrames (float division).
*DataFrame.truediv* Divide DataFrames (float division).
*DataFrame.floordiv* Divide DataFrames (integer division).
*DataFrame.mod* Calculate modulo (remainder after division).
*DataFrame.pow* Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                   'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
          angles  degrees
     circle      0     360
    triangle      3     180
rectangle      4     360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
          angles  degrees
     circle      1     361
    triangle      4     181
rectangle      5     361
```

```python
>>> df.add(1)
          angles  degrees
     circle      1     361
    triangle      4     181
rectangle      5     361
```

Divide by constant with reverse version.
```python
>>> df.div(10)
   angles  degrees
circle  0.0     36.0
triangle 0.3     18.0
rectangle 0.4     36.0

>>> df.rdiv(10)
   angles  degrees
circle  inf     0.027778
triangle 3.33333  0.055556
rectangle 2.50000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
   angles  degrees
circle -1     358
triangle 2     178
rectangle 3     358

>>> df.sub([1, 2], axis='columns')
   angles  degrees
circle -1     358
triangle 2     178
rectangle 3     358

>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')
   angles  degrees
circle -1     359
triangle 2     179
rectangle 3     359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
>>> other
     angles
circle     0
triangle    3
rectangle   4

>>> df * other
   angles  degrees
circle    0     NaN
triangle   9     NaN
rectangle 16     NaN

>>> df.mul(other, fill_value=0)
   angles  degrees
circle    0     0.0
```

(continues on next page)
triangle 9 0.0
rectangle 16 0.0

Divide by a MultiIndex by level.

```python
def multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
...                      'degrees': [360, 180, 360, 540, 720],
...               index=['A', 'A', 'A', 'B', 'B'],
...                     ['circle', 'triangle', 'rectangle',
...                      'square', 'pentagon', 'hexagon']})
```

```python
def div(df_multindex, level=1, fill_value=0)
a
A circle 0 360
triangle 3 180
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720
```
**random_state** [int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

   Changed in version 1.1.0: array-like and BitGenerator object now passed to np.random.RandomState() as seed

   Changed in version 1.4.0: np.random.Generator objects now accepted

**axis** [[0 or ‘index’, 1 or ‘columns’, None], default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames).

**ignore_index** [bool, default False] If True, the resulting index will be labeled 0, 1, …, n - 1.

   New in version 1.3.0.

**Returns**

   Series or DataFrame A new object of same type as caller containing \( n \) items randomly sampled from the caller object.

**See also:**

**DataFrameGroupBy.sample** Generates random samples from each group of a DataFrame object.

**SeriesGroupBy.sample** Generates random samples from each group of a Series object.

**numpy.random.choice** Generates a random sample from a given 1-D numpy array.

**Notes**

   If \( frac > 1 \), replacement should be set to True.

**Examples**

```python
>>> df = pd.DataFrame({
    'num_legs': [2, 4, 8, 0],
    'num_wings': [2, 0, 0, 0],
    'num_specimen_seen': [10, 2, 1, 8],
    'index': ['falcon', 'dog', 'spider', 'fish']
})

>>> df
  num_legs  num_wings  num_specimen_seen
falcon      2          2               10
  dog        4          0               2
spider      8          0               1
  fish       0          0               8

Extract 3 random elements from the Series df['num_legs']: Note that we use random_state to ensure the reproducibility of the examples.

```
A random 50% sample of the DataFrame with replacement:

```python
>>> df.sample(frac=0.5, replace=True, random_state=1)
   num_legs  num_wings  num_specimen_seen
dog       4          0            2
fish      0          0            8
```

An upsample sample of the DataFrame with replacement: Note that replace parameter has to be True for frac parameter > 1.

```python
>>> df.sample(frac=2, replace=True, random_state=1)
   num_legs  num_wings  num_specimen_seen
  dog       4          0            2
 fish      0          0            8
falcon     2          2           10
 falcon     2          2           10
  fish      0          0            8
  dog       4          0            2
  fish      0          0            8
  dog       4          0            2
```

Using a DataFrame column as weights. Rows with larger value in the num_specimen_seen column are more likely to be sampled.

```python
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
   num_legs  num_wings  num_specimen_seen
falcon     2          2           10
 fish      0          0            8
```

**pandas.DataFrame.select_dtypes**

DataFrame.select_dtypes(include=None, exclude=None)

Return a subset of the DataFrame’s columns based on the column dtypes.

**Parameters**

- **include, exclude** [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

**Returns**

DataFrame The subset of the frame including the dtypes in include and excluding the dtypes in exclude.

**Raises**

- ValueError
  - If both of include and exclude are empty
  - If include and exclude have overlapping elements
  - If any kind of string dtype is passed in.

**See also:**

- DataFrame.dtypes Return Series with the data type of each column.
Notes

- To select all numeric types, use `np.number` or 'number'
- To select strings you must use the object dtype, but note that this will return all object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, 'datetime' or 'datetime64'
- To select timedeltas, use `np.timedelta64`, 'timedelta' or 'timedelta64'
- To select Pandas categorical dtypes, use 'category'
- To select Pandas datetimetz dtypes, use 'datetimetz' (new in 0.20.0) or 'datetime64[ns, tz]'

Examples

```python
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
   a  b  c
0  1  True  1.0
1  2  False  2.0
2  1  True  1.0
3  2  False  2.0
4  1  True  1.0
5  2  False  2.0

>>> df.select_dtypes(include='bool')
       b
0    True
1   False
2    True
3   False
4    True
5   False

>>> df.select_dtypes(include=['float64'])
    c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0

>>> df.select_dtypes(exclude=['int64'])
   b  c
0  True  1.0
1  False  2.0
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>True</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.sem**

DataFrame.sem(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased standard error of the mean over requested axis. Normalized by N-1 by default. This can be changed using the ddof argument

**Parameters**

- **axis** ([index (0), columns (1)])
- **skipna** (bool, default True) Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- **numeric_only** (bool, default None) Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

Series or DataFrame (if level specified)

**pandas.DataFrame.set_axis**

DataFrame.set_axis(labels, axis=0, inplace=False)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

**Parameters**

- **labels** [list-like, Index] The values for the new index.
- **axis** ([0 or ‘index’, 1 or ‘columns’], default 0) The axis to update. The value 0 identifies the rows, and 1 identifies the columns.
- **inplace** (bool, default False) Whether to return a new DataFrame instance.

**Returns**

renamed [DataFrame or None] An object of type DataFrame or None if inplace=True.

See also:

- **DataFrame.rename_axis** Alter the name of the index or columns.
Examples

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```python
>>> df.set_axis(['a', 'b', 'c'], axis='index')
```

```
A  B
a 1 4
b 2 5
c 3 6
```

Change the column labels.

```python
>>> df.set_axis(['I', 'II'], axis='columns')
```

```
I  II
0 1 4
1 2 5
2 3 6
```

Now, update the labels inplace.

```python
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
```

```
i  ii
0 1 4
1 2 5
2 3 6
```

`pandas.DataFrame.set_flags`

`DataFrame.set_flags(*, copy=False, allows_duplicate_labels=None)`

Return a new object with updated flags.

- **Parameters**
  - `allows_duplicate_labels` [bool, optional] Whether the returned object allows duplicate labels.

- **Returns**
  - `Series` or `DataFrame` The same type as the caller.

**See also:**

- `DataFrame.attrs` Global metadata applying to this dataset.
- `DataFrame.flags` Global flags applying to this object.
Notes

This method returns a new object that’s a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

“Flags” differ from “metadata”. Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in DataFrame.attrs.

Examples

```python
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allow_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

pandas.DataFrame.set_index

DataFrame.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

Parameters

- **keys** [label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, “array” encompasses Series, Index, np.ndarray, and instances of Iterator.

- **drop** [bool, default True] Delete columns to be used as the new index.

- **append** [bool, default False] Whether to append columns to existing index.

- **inplace** [bool, default False] If True, modifies the DataFrame in place (do not create a new object).

- **verify_integrity** [bool, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method.

Returns

- DataFrame or None Changed row labels or None if inplace=True.

See also:

- DataFrame.reset_index Opposite of set_index.
- DataFrame.reindex Change to new indices or expand indices.
- DataFrame.reindex_like Change to same indices as other DataFrame.
Examples

```python
>>> df = pd.DataFrame({
    'month': [1, 4, 7, 10],
    'sale': [55, 40, 84, 31])
```

```python
df
```
```
## month  year  sale
---  ---  ----
 0  1   2012 55
 1  4   2014 40
 2  7   2013 84
 3 10   2014 31
```

Set the index to become the ‘month’ column:

```python
>>> df.set_index('month')
```

```
## year  sale
---  ----
 1  2012 55
 4  2014 40
 7  2013 84
10  2014 31
```

Create a MultiIndex using columns ‘year’ and ‘month’:

```python
>>> df.set_index(['year', 'month'])
```

```
## sale
 year  month
---  ---
2012  1  55
2014  4  40
2013  7  84
2014 10  31
```

Create a MultiIndex using an Index and a column:

```python
>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
```

```
## month  sale
 year  ---
1  2012 1  55
2  2014 4  40
3  2013 7  84
4  2014 10 31
```

Create a MultiIndex using two Series:

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
```

```
## month  year  sale
 year  ---  ----
1  1  2012 55
2  4  2014 40
3  9  2013 84
4 16  2014 31
```
pandas.DataFrame.shift

DataFrame.shift( periods=1, freq=None, axis=0, fill_value=NoDefault.no_default )
Shift index by desired number of periods with an optional time freq.

When freq is not passed, shift the index without realigning the data. If freq is passed (in this case, the index must be date or datetime, or it will raise a NotImplmentedError), the index will be increased using the periods and the freq. freq can be inferred when specified as “infer” as long as either freq or inferred_freq attribute is set in the index.

Parameters

periods [int] Number of periods to shift. Can be positive or negative.

freq [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. ‘EOM’). If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data. If freq is specified as “infer” then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

axis [[0 or ‘index’, 1 or ‘columns’, None], default None] Shift direction.

fill_value [object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of self. For numeric data, np.nan is used. For datetime, timedelta, or period data, etc. NaT is used. For extension dtypes, self. dtype.na_value is used.

Changed in version 1.1.0.

Returns

DataFrame Copy of input object, shifted.

See also:

Index.shift Shift values of Index.

DatetimeIndex.shift Shift values of DatetimeIndex.

PeriodIndex.shift Shift values of PeriodIndex.

tshift Shift the time index, using the index’s frequency if available.

Examples

```python
>>> df = pd.DataFrame( { "Col1": [10, 20, 15, 30, 45], "Col2": [13, 23, 18, 33, 48], "Col3": [17, 27, 22, 37, 52] }
... index=pd.date_range("2020-01-01", "2020-01-05")
)      
>>> df
     Col1  Col2  Col3
2020-01-01  10   13   17
2020-01-02  20   23   27
2020-01-03  15   18   22
2020-01-04  30   33   37
2020-01-05  45   48   52
```
```python
>>> df.shift(periods=3)
Col1  Col2  Col3
2020-01-01 NaN  NaN  NaN
2020-01-02 NaN  NaN  NaN
2020-01-03 NaN  NaN  NaN
2020-01-04 10.0 13.0 17.0
2020-01-05 20.0 23.0 27.0
```

```python
>>> df.shift(periods=1, axis="columns")
Col1  Col2  Col3
2020-01-01 NaN  10   13
2020-01-02 NaN  20   23
2020-01-03 NaN  15   18
2020-01-04 NaN  30   33
2020-01-05 NaN  45   48
```

```python
>>> df.shift(periods=3, fill_value=0)
Col1  Col2  Col3
2020-01-01 0    0    0
2020-01-02 0    0    0
2020-01-03 0    0    0
2020-01-04 10   13   17
2020-01-05 20   23   27
```

```python
>>> df.shift(periods=3, freq="D")
Col1  Col2  Col3
2020-01-04 10   13   17
2020-01-05 20   23   27
2020-01-06 15   18   22
2020-01-07 30   33   37
2020-01-08 45   48   52
```

```python
>>> df.shift(periods=3, freq="infer")
Col1  Col2  Col3
2020-01-04 10   13   17
2020-01-05 20   23   27
2020-01-06 15   18   22
2020-01-07 30   33   37
2020-01-08 45   48   52
```

**pandas.DataFrame.skew**

DataFrame.skew(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None, **kwargs)

Return unbiased skew over requested axis.

Normalized by N-1.

**Parameters**

- **axis** ([index (0), columns (1)]) Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
The `DataFrame` class in pandas provides methods for slicing and sorting:

### `pandas.DataFrame.slice_shift`:

*Slice shift method*

```python
def slice_shift(periods=1, axis=0)
```

Equivalent to `shift` without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

**Parameters**
- `periods` [int] Number of periods to move, can be positive or negative.

**Returns**
- `shifted` [same type as caller]

**Notes**

While the `slice_shift` is faster than `shift`, you may pay for it later during alignment.

### `pandas.DataFrame.sort_index`:

*Sort index method*:

```python
def sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)
```

Sort object by labels (along an axis).

**Returns**
- A new DataFrame sorted by label if `inplace` argument is `False`, otherwise updates the original DataFrame and returns `None`.

**Parameters**
- `axis` [[0 or ‘index’, 1 or ‘columns’], default 0] The axis along which to sort. The value 0 identifies the rows, and 1 identifies the columns.
- `level` [int or level name or list of ints or list of level names] If not None, sort on values in specified index level(s).
- `ascending` [bool or list-like of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.
- `inplace` [bool, default False] If True, perform operation in-place.
kind [‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’], default ‘quicksort’] Choice of sorting algorithm. See also numpy.sort() for more information. mergesort and stable are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{‘first’, ‘last’}, default ‘last’] Puts NaNs at the beginning if first; last puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

ignore_index [bool, default False] If True, the resulting axis will be labeled 0, 1, ..., n - 1.

New in version 1.0.0.

key [callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the key argument in the builtin sorted() function, with the notable difference that this key function should be vectorized. It should expect an Index and return an Index of the same shape. For MultiIndex inputs, the key is applied per level.

New in version 1.1.0.

Returns

DataFrame or None The original DataFrame sorted by the labels or None if inplace=True.

See also:

Series.sort_index Sort Series by the index.

DataFrame.sort_values Sort DataFrame by the value.

Series.sort_values Sort Series by the value.

Examples

```python
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
                    columns=['A'])
>>> df.sort_index()
   A
1  4
29 2
100 1
150 5
234 3
```

By default, it sorts in ascending order, to sort in descending order, use ascending=False

```python
>>> df.sort_index(ascending=False)
   A
234 3
150 5
100 1
29 2
1  4
```
A key function can be specified which is applied to the index before sorting. For a MultiIndex this is applied to each level separately.

```python
>>> df = pd.DataFrame({'a': [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
>>> df.sort_index(key=lambda x: x.str.lower())
    a
A  1
b  2
C  3
d  4
```

**pandas.DataFrame.sort_values**

DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)

Sort by the values along either axis.

**Parameters**

- **by** [str or list of str] Name or list of names to sort by.
  - If `axis` is 0 or 'index' then `by` may contain index levels and/or column labels.
  - If `axis` is 1 or 'columns' then `by` may contain column levels and/or index labels.

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted.

- **ascending** [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

- **inplace** [bool, default False] If True, perform operation in-place.

- **kind** [{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also numpy.sort() for more information. mergesort and stable are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

- **na_position** [{‘first’, ‘last’}, default ‘last’] Puts NaNs at the beginning if first; last puts NaNs at the end.

- **ignore_index** [bool, default False] If True, the resulting axis will be labeled 0, 1, .., n - 1.
  
  New in version 1.0.0.

- **key** [callable, optional] Apply the key function to the values before sorting. This is similar to the key argument in the builtin sorted() function, with the notable difference that this key function should be vectorized. It should expect a Series and return a Series with the same shape as the input. It will be applied to each column in by independently.
  
  New in version 1.1.0.

**Returns**

- **DataFrame or None** DataFrame with sorted values or None if inplace=True.

**See also:**

- **DataFrame.sort_index** Sort a DataFrame by the index.
Series.sort_values  Similar method for a Series.

Examples

```python
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
```
```python
>>> df
   col1 col2 col3 col4
0    A    2    0    a
1    A    1    1    B
2    B    9    9    c
3  NaN    8    4    D
4    D    7    2    e
5    C    4    3    F
```
Sort by col1
```python
>>> df.sort_values(by=['col1'])
   col1 col2 col3 col4
0    A    2    0    a
1    A    1    1    B
2    B    9    9    c
5    C    4    3    F
4    D    7    2    e
3  NaN    8    4    D
```
Sort by multiple columns
```python
>>> df.sort_values(by=['col1', 'col2'])
   col1 col2 col3 col4
1    A    1    1    B
0    A    2    0    a
2    B    9    9    c
5    C    4    3    F
4    D    7    2    e
3  NaN    8    4    D
```
Sort Descending
```python
>>> df.sort_values(by='col1', ascending=False)
   col1 col2 col3 col4
4    D    7    2    e
5    C    4    3    F
2    B    9    9    c
0    A    2    0    a
1    A    1    1    B
3  NaN    8    4    D
```
Putting NAs first

3.4. DataFrame
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.sort_values(by='col1', ascending=False, na_position='first')
col1  col2  col3  col4
3   NaN  8   4   D
4    D  7   2   e
5    C  4   3   F
2    B  9   9   c
0    A  2   0   a
1    A  1   1   B
```

Sorting with a key function

```python
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
col1  col2  col3  col4
0    A  2   0   a
1    A  1   1   B
2    B  9   9   c
3   NaN  8   4   D
4    D  7   2   e
5    C  4   3   F
```

Natural sort with the key argument, using the natsort <https://github.com/SethMMorton/natsort> package.

```python
>>> df = pd.DataFrame({
...   "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...   "value": [10, 20, 30, 40, 50]
... })
>>> df
   time  value
0  0hr     10
1 128hr    20
2  72hr    30
3  48hr    40
4  96hr    50
>>> from natsort import index_natsorted
>>> df.sort_values(
...   by="time",
...   key=lambda x: np.argsort(index_natsorted(df["time"])))
   time  value
0  0hr     10
1  48hr    40
2  72hr    30
3  96hr    50
4 128hr    20
```
pandas.DataFrame.sparse

DataFrame.sparse()
DataFrame accessor for sparse data.
New in version 0.25.0.

pandas.DataFrame.squeeze

DataFrame.squeeze(axis=None)
Squeeze 1 dimensional axis objects into scalars.
Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.
This method is most useful when you don’t know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call squeeze to ensure you have a Series.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

Returns

DataFrame, Series, or scalar The projection after squeezing axis or all the axes.

See also:

Series.iloc Integer-location based indexing for selecting scalars.
DataFrame.iloc Integer-location based indexing for selecting Series.
Series.to_frame Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

>>> primes = pd.Series([2, 3, 5, 7])
Slicing might produce a Series with a single value:

>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0   2
dtype: int64

>>> even_primes.squeeze()
2
Squeezing objects with more than one value in every axis does nothing:

>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1  3
2  5
(continues on next page)
Squeezing is even more effective when used with DataFrames.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```python
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```python
>>> df_a.squeeze('columns')
0  1
1  3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```python
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```python
>>> df_0a.squeeze('rows')
a  1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```python
>>> df_0a.squeeze()
1
```
DataFrame.stack

DataFrame.stack(level=-1, dropna=True)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

• if the columns have a single level, the output is a Series;
• if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

Parameters

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

Returns

DataFrame or Series  Stacked dataframe or series.

See also:

DataFrame.unstack  Unstack prescribed level(s) from index axis onto column axis.
DataFrame.pivot  Reshape dataframe from long format to wide format.
DataFrame.pivot_table  Create a spreadsheet-style pivot table as a DataFrame.

Notes

The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

Reference the user guide for more examples.

Examples

Single level columns

```python
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
          index=['cat', 'dog'],
          columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:
```python
>>> df_single_level_cols
    weight  height
   cat    0      1
   dog    2      3
>>> df_single_level_cols.stack()
   weight
  cat    0
     height
   cat    1
   dog    2
   height 3
dtype: int64
```

Multi level columns: simple case

```python
>>> multicoll = pd.MultiIndex.from_tuples(
    [('weight', 'kg'), ... 
     ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]], ...
   index=['cat', 'dog'], ...
   columns=multicoll)
```

Stacking a dataframe with a multi-level column axis:

```python
>>> df_multi_level_cols1
    weight
   kg  pounds
  cat    1      2
  dog    2      4
>>> df_multi_level_cols1.stack()
    weight
   cat kg     1
         pounds
   cat              2
   dog kg     2
         pounds
   dog              4
```

Missing values

```python
>>> multicol2 = pd.MultiIndex.from_tuples(
    [('weight', 'kg'), ... 
     ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]], ...
   index=['cat', 'dog'], ...
   columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```python
>>> df_multi_level_cols2
    weight  height
       kg     m
   cat  1.0   2.0
   dog  3.0   4.0
>>> df_multi_level_cols2.stack()
    height
   weight
  cat kg   NaN  1.0
         m     2.0
   dog kg   NaN  NaN
```
Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
   kg   m
cat height NaN  2.0
      weight 1.0  NaN
dog height NaN  4.0
      weight 3.0  NaN
```

```
>>> df_multi_level_cols2.stack([0, 1])
   cat height m  2.0
           weight kg  1.0
   dog height m  4.0
           weight kg  3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
                           index=['cat', 'dog'],
                           columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
height weight
kg m
cat NaN 1.0
dog 2.0 3.0
```

```
>>> df_multi_level_cols3.stack(dropna=False)
height weight
cat kg NaN NaN
      m  1.0 NaN
dog kg NaN  2.0
      m  3.0 NaN
```

```
>>> df_multi_level_cols3.stack(dropna=True)
height weight
cat m  1.0 NaN
dog kg NaN  2.0
      m  3.0 NaN
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DataFrame.std**

DataFrame.

```
DataFrame.
```

```
std

(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)
```

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

**Parameters**

- **axis** [{index (0), columns (1)}]
- **skipna** [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- **numeric_only** [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

Series or DataFrame (if level specified)

**Notes**

To have the same behaviour as `numpy.std`, use `ddof=0` (instead of the default `ddof=1`)

**Examples**

```python
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...    'age': [21, 25, 62, 43],
...    'height': [1.61, 1.87, 1.49, 2.01]}
...   ).set_index('person_id')
```

```
<table>
<thead>
<tr>
<th>person_id</th>
<th>age</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21</td>
<td>1.61</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>1.87</td>
</tr>
<tr>
<td>2</td>
<td>62</td>
<td>1.49</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>2.01</td>
</tr>
</tbody>
</table>
```

The standard deviation of the columns can be found as follows:

```python
>>> df.std()
```

```
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>18.786076</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>0.237417</td>
<td></td>
</tr>
</tbody>
</table>
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:
```python
>>> df.std(ddof=0)
age    16.269219
height  0.205609
```

**pandas.DataFrame.sub**

`DataFrame.sub(other, axis='columns', level=None, fill_value=None)`

Get Subtraction of dataframe and other, element-wise (binary operator sub).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, `rsub`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

- **DataFrame** Result of the arithmetic operation.

**See also:**

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360],
...                    index=['circle', 'triangle', 'rectangle'])
>>> df
   angles  degrees
circle     0      360
triangle   3      180
rectangle  4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
   angles  degrees
circle     1      361
triangle   4      181
rectangle  5      361
```

```python
>>> df.add(1)
   angles  degrees
circle     1      361
triangle   4      181
rectangle  5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
   angles  degrees
circle  0.0     36.0
triangle 0.3     18.0
rectangle 0.4     36.0
```

```python
>>> df.rdiv(10)
   angles  degrees
circle   inf      0.027778
triangle  3.333333 0.055556
rectangle  2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
   angles  degrees
circle  -1      358
triangle   2      178
rectangle  3      358
```
Multiply a DataFrame of different shape with operator version.

```python
def * other
circle 0  NaN
triangle 9  NaN
rectangle 16 NaN
```

Divide by a MultiIndex by level.

```python
def_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6],
                             'degrees': [360, 180, 360, 540, 720],
                             'index': ['A', 'A', 'A', 'B', 'B'],
                             'column': ['circle', 'triangle', 'rectangle',
                                       'square', 'pentagon', 'hexagon']})
```
```python
>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
   A  circle  NaN   1.0
       triangle  1.0   1.0
       rectangle  1.0   1.0
   B  square  0.0   0.0
       pentagon  0.0   0.0
       hexagon  0.0   0.0
```

**pandas.DataFrame.subtract**

`DataFrame.subtract(other, axis='columns', level=None, fill_value=None)`

Get Subtraction of dataframe and other, element-wise (binary operator `sub`).

Equivalent to `dataframe - other`, but with support to substitute a `fill_value` for missing data in one of the inputs. With reverse version, `rsub`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- `level` [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- `fill_value` [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

`DataFrame` Result of the arithmetic operation.

**See also:**

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360],
...                     index=['circle', 'triangle', 'rectangle'])
>>> df
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
      angles  degrees
circle     1     361
triangle    4     181
rectangle   5     361
```

```python
>>> df.add(1)
      angles  degrees
circle     1     361
triangle    4     181
rectangle   5     361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
      angles  degrees
circle   0.0    36.0
triangle 0.3    18.0
rectangle 0.4    36.0
```

```python
>>> df.rdiv(10)
      angles  degrees
circle  inf   0.027778
triangle 3.333333  0.055556
rectangle 2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
      angles  degrees
circle    -1    358
triangle     2    178
rectangle    3    358
```
Multiplying and dividing a DataFrame

The behavior of elementwise operations may be more intuitive when specifying the axes:

```python
In [1]: df = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                      'degrees': [360, 180, 360, 360, 540, 720],
                      'index': ['A', 'A', 'A', 'B', 'B', 'B']},
                      index=[['circle', 'triangle', 'rectangle'],
                              'square', 'pentagon', 'hexagon'])

In [2]: df
Out[2]:
   angles  degrees
   index
  A circle   0    360
  A triangle  3    180
  A rectangle  4    360
  B square   4    360
  B pentagon  5    540
  B hexagon  6    720

In [3]: df.mul(other, fill_value=0)
   angles  degrees
   index
  A circle   0     0.0
  A triangle  9     0.0
  A rectangle 16     0.0

Dividing by a MultiIndex by level.

```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.div(df_multindex, level=1, fill_value=0)
   angles  degrees
A  circle  NaN   1.0
    triangle   1.0   1.0
    rectangle   1.0   1.0
B   square   0.0   0.0
    pentagon   0.0   0.0
    hexagon   0.0   0.0
```

**pandas.DataFrame.sum**

`DataFrame.sum(axis=None, skipna=True, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the sum of the values over the requested axis.

This is equivalent to the method `numpy.sum`.

**Parameters**

- `axis` [[index (0), columns (1)]] Axis for the function to be applied on.
- `skipna` [bool, default True] Exclude NA/null values when computing the result.
- `level` [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- `numeric_only` [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- `min_count` [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.
- `**kwargs` Additional keyword arguments to be passed to the function.

**Returns**

- `Series` or `DataFrame` (if level specified)

**See also:**

- `Series.sum` Return the sum.
- `Series.min` Return the minimum.
- `Series.max` Return the maximum.
- `Series.idxmin` Return the index of the minimum.
- `Series.idxmax` Return the index of the maximum.
- `DataFrame.sum` Return the sum over the requested axis.
- `DataFrame.min` Return the minimum over the requested axis.
- `DataFrame.max` Return the maximum over the requested axis.
- `DataFrame.idxmin` Return the index of the minimum over the requested axis.
- `DataFrame.idxmax` Return the index of the maximum over the requested axis.
Examples

```python
>>> idx = pd.MultiIndex.from_arrays(
...     [['warm', 'warm', 'cold', 'cold'],
...      ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
  warm  dog  4
     falcon  2
cold  fish  0
    spider  8
Name: legs, dtype: int64
```

```python
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is 0.

```python
>>> pd.Series([], dtype='float64').sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```python
>>> pd.Series([], dtype='float64').sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).sum()
0.0
```

```python
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

### pandas.DataFrame.swapaxes

**DataFrame.swapaxes(axis1, axis2, copy=True)**

Interchange axes and swap values axes appropriately.

**Returns**

- `y` [same as input]
DataFrame.swaplevel

DataFrame.swaplevel(i=-2, j=-1, axis=0)

Swap levels i and j in a MultiIndex.
Default is to swap the two innermost levels of the index.

Parameters

i, j [int or str] Levels of the indices to be swapped. Can pass level name as string.
axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

Returns

DataFrame DataFrame with levels swapped in MultiIndex.

Examples

```python
>>> df = pd.DataFrame({
    "Grade": ["A", "B", "A", "C"],
    "index": [
        ["Final exam", "Final exam", "Coursework", "Coursework"],
        ["History", "Geography", "History", "Geography"],
        ["January", "February", "March", "April"],
    ]
}).
```

In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```python
>>> df.swaplevel()
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```python
>>> df.swaplevel(0)
```
We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```python
>>> df.swaplevel(0, 1)
Grade
History  Final exam  January  A
Geography Final exam  February  B
History  Coursework  March  A
Geography Coursework  April  C
```

### pandas.DataFrame.tail

**DataFrame.tail(n=5)**

Return the last $n$ rows.

This function returns last $n$ rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of $n$, this function returns all rows except the first $n$ rows, equivalent to `df[n:]`.

**Parameters**

- **n** [int, default 5] Number of rows to select.

**Returns**

- **type of caller** The last $n$ rows of the caller object.

**See also:**

- `DataFrame.head` The first $n$ rows of the caller object.

**Examples**

```python
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
... 'monkey', 'parrot', 'shark', 'whale', 'zebra']})
```

>>> df

<table>
<thead>
<tr>
<th></th>
<th>animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alligator</td>
</tr>
<tr>
<td>1</td>
<td>bee</td>
</tr>
<tr>
<td>2</td>
<td>falcon</td>
</tr>
<tr>
<td>3</td>
<td>lion</td>
</tr>
<tr>
<td>4</td>
<td>monkey</td>
</tr>
<tr>
<td>5</td>
<td>parrot</td>
</tr>
<tr>
<td>6</td>
<td>shark</td>
</tr>
<tr>
<td>7</td>
<td>whale</td>
</tr>
<tr>
<td>8</td>
<td>zebra</td>
</tr>
</tbody>
</table>

Viewing the last 5 lines

```python
>>> df.tail()
```

<table>
<thead>
<tr>
<th></th>
<th>animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>monkey</td>
</tr>
<tr>
<td>5</td>
<td>parrot</td>
</tr>
</tbody>
</table>

(continues on next page)
Viewing the last \( n \) lines (three in this case)

```python
>>> df.tail(3)
   animal
   6  shark
   7  whale
   8  zebra
```

For negative values of \( n \)

```python
>>> df.tail(-3)
   animal
   3   lion
   4  monkey
   5  parrot
   6  shark
   7  whale
   8  zebra
```

**pandas.DataFrame.take**

`DataFrame.take(indices, axis=0, is_copy=None, **kwargs)`

Return the elements in the given positional indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters**

- `indices` [array-like] An array of ints indicating which positions to take.
- `axis` [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.
- `is_copy` [bool] Before pandas 1.0, `is_copy=False` can be specified to ensure that the return value is an actual copy. Starting with pandas 1.0, `take` always returns a copy, and the keyword is therefore deprecated.

Deprecated since version 1.0.0.

- `**kwargs` For compatibility with `numpy.take()`. Has no effect on the output.

**Returns**

- `taken` [same type as caller] An array-like containing the elements taken from the object.

**See also:**

- `DataFrame.loc` Select a subset of a DataFrame by labels.
- `DataFrame.iloc` Select a subset of a DataFrame by positions.
- `numpy.take` Take elements from an array along an axis.
Examples

```python
>>> df = pd.DataFrame([(
'falcon', 'bird', 389.0),
... ('parrot', 'bird', 24.0),
... ('lion', 'mammal', 80.5),
... ('monkey', 'mammal', np.nan),
... columns=['name', 'class', 'max_speed'],
... index=[0, 2, 3, 1])
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```python
>>> df.take([0, 3])
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```python
>>> df.take([1, 2], axis=1)
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```python
>>> df.take([-1, -2])
```

**pandas.DataFrame.to_dict**

DataFrame.to_dict(orient='dict', into=<class 'dict'>)

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

**Parameters**

- **orient** [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.
  - `dict` (default) : dict like {column -> {index -> value}}
• ‘list’ : dict like {column -> [values]}
• ‘series’ : dict like {column -> Series(values)}
• ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
• ‘tight’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values],
  ‘index_names’ -> [index.names], ‘column_names’ -> [column.names]}
• ‘records’ : list like [{column -> value}, … , {column -> value}]
• ‘index’ : dict like {index -> {column -> value}}

Abbreviations are allowed. s indicates series and sp indicates split.

New in version 1.4.0: ‘tight’ as an allowed value for the orient argument

Returns

dict, list or collections.abc.Mapping  Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the orient parameter.

See also:

DataFrame.from_dict  Create a DataFrame from a dictionary.
DataFrame.to_json  Convert a DataFrame to JSON format.

Examples

```python
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                    index=['row1', 'row2'])
```

```python
>>> df
   col1  col2
row1  1   0.50
row2  2   0.75
```

```python
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```python
>>> df.to_dict('series')
{'col1': row1 1
  row2 2
Name: col1, dtype: int64,
'col2': row1 0.50
  row2 0.75
Name: col2, dtype: float64}
```

```python
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df.to_dict('records')
[['col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]

>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}]

>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

You can also specify the mapping type.

```python
>>> from collections import OrderedDict, defaultdict

>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])

If you want a `defaultdict`, you need to initialize it:

```python
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

---

**pandas.DataFrame.to_gbq**

`DataFrame.to_gbq(destination_table, project_id=None, chunksize=None, reauth=False, if_exists='fail', auth_local_webserver=False, table_schema=None, location=None, progress_bar=True, credentials=None)`

Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

See the [How to authenticate with Google BigQuery](#) guide for authentication instructions.

**Parameters**

- **destination_table** [str] Name of table to be written, in the form dataset.tablename.
- **project_id** [str, optional] Google BigQuery Account project ID. Optional when available from the environment.
- **chunksize** [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to None to load the whole dataframe at once.
- **reauth** [bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.
- **if_exists** [str, default ‘fail’] Behavior when the destination table exists. Value can be one of:
  - ‘fail’ If table exists raise pandas_gbq.gbq.TableCreationError.
  - ‘replace’ If table exists, drop it, recreate it, and insert data.
  - ‘append’ If table exists, insert data. Create if does not exist.
auth_local_webserver  [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

*New in version 0.2.0 of pandas-gbq.*

table_schema  [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{"name": 'col1', 'type': 'STRING'},{...}]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

*New in version 0.3.1 of pandas-gbq.*

location  [str, optional] Location where the load job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of the target dataset.

*New in version 0.5.0 of pandas-gbq.*

progress_bar  [bool, default True] Use the library tqdm to show the progress bar for the upload, chunk by chunk.

*New in version 0.5.0 of pandas-gbq.*

credentials  [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

*New in version 0.8.0 of pandas-gbq.*

See also:

- `pandas_gbq.to_gbq` This function in the pandas-gbq library.
- `read_gbq` Read a DataFrame from Google BigQuery.

**pandas.DataFrame.to_hdf**

`DataFrame.to_hdf(path_or_buf, key, mode='a', complevel=None, complib=None, append=False, format=None, index=True, min_itemsize=None, nan_rep=None, dropna=None, data_columns=None, errors='strict', encoding='UTF-8')`

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

**Warning:** One can store a subclass of DataFrame or Series to HDF5, but the type of the subclass is lost upon storing.

For more information see the user guide.

**Parameters**

- path_or_buf  [str or pandas.HDFStore] File path or HDFStore object.
**key** [str] Identifier for the group in the store.

**mode** [{‘a’, ‘w’, ‘r+’}, default ‘a’] Mode to open file:
- ‘w’: write, a new file is created (an existing file with the same name would be deleted).
- ‘a’: append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- ‘r+’: similar to ‘a’, but the file must already exist.

**complevel** [{0-9}, default None] Specifies a compression level for data. A value of 0 or None disables compression.

**complib** [{‘zlib’, ‘lz4’, ‘bzip2’, ‘blosc’}, default ‘zlib’] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: ‘blosc:blosclz’): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd’}. Specifying a compression library which is not available issues a ValueError.

**append** [bool, default False] For Table formats, append the input data to the existing.

**format** [{‘fixed’, ‘table’, None}, default ‘fixed’] Possible values:
- ‘table’: Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.
- If None, pd.get_option(‘io.hdf.default_format’) is checked, followed by fallback to “fixed”.

**errors** [str, default ‘strict’] Specifies how encoding and decoding errors are to be handled. See the errors argument for open() for a full list of options.

**encoding** [str, default “UTF-8”]

**min_itemsize** [dict or int, optional] Map column names to minimum string sizes for columns.

**nan_rep** [Any, optional] How to represent null values as str. Not allowed with append=True.

**data_columns** [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See Query via data columns. Applicable only to format=‘table’.

See also:

*read_hdf* Read from HDF file.

*DataFrame.to_parquet* Write a DataFrame to the binary parquet format.

*DataFrame.to_sql* Write to a SQL table.

*DataFrame.to_feather* Write out feather-format for DataFrames.

*DataFrame.to_csv* Write out to a csv file.
Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')

We can add another object to the same file:

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')

Reading from HDF file:

```python
>>> pd.read_hdf('data.h5', 'df')
A  B
a 1 4
b 2 5
c 3 6

>>> pd.read_hdf('data.h5', 's')
0 1
1 2
2 3
3 4
dtype: int64
```

`pandas.DataFrame.to_markdown`

`DataFrame.to_markdown(buf=None, mode='wt', index=True, storage_options=None, **kwargs)`

Print DataFrame in Markdown-friendly format.

New in version 1.0.0.

Parameters

- **buf** [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.
- **mode** [str, optional] Mode in which file is opened, “wt” by default.
- **index** [bool, optional, default True] Add index (row) labels.

New in version 1.1.0.

- **storage_options** [dict, optional] Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with “s3://”, or “gcs://”) the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

New in version 1.2.0.

- **kwargs** These parameters will be passed to `tabulate`.

Returns

- **str** DataFrame in Markdown-friendly format.
Notes

Requires the `tabulate` package.

Examples

```python
>>> df = pd.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(df.to_markdown())
<table>
<thead>
<tr>
<th></th>
<th>animal_1</th>
<th>animal_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>elk</td>
<td>dog</td>
</tr>
<tr>
<td>1</td>
<td>pig</td>
<td>quetzal</td>
</tr>
</tbody>
</table>
```

Output markdown with a tabulate option.

```python
>>> print(df.to_markdown(tablefmt="grid"))
+----+------------+------------+
|    | animal_1   | animal_2   |
+====+============+============+
| 0  | elk        | dog        |
+----+------------+------------+
| 1  | pig        | quetzal    |
+----+------------+------------+
```

`pandas.DataFrame.to_numpy`

DataFrame.to_numpy(dtype=None, copy=False, na_value=NoDefault.no_default)

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`.
copy [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not ensure that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.
na_value [Any, optional] The value to use for missing values. The default value depends on `dtype` and the dtypes of the DataFrame columns.

New in version 1.1.0.

Returns

numpy.ndarray

See also:

`Series.to_numpy` Similar method for Series.
Examples

```python
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```python
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
```

```python
>>> df.to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```python
>>> df['C'] = pd.date_range('2000', periods=2)
```

```python
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

```
pandas.DataFrame.to_period
```

DataFrame.to_period(freq=None, axis=0, copy=True)

Convert DataFrame from DatetimeIndex to PeriodIndex.

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

- freq [str, default] Frequency of the PeriodIndex.
- axis [{0 or 'index', 1 or 'columns'}, default=0] The axis to convert (the index by default).
- copy [bool, default True] If False then underlying input data is not copied.

Returns

DataFrame with PeriodIndex

Examples

```python
>>> idx = pd.to_datetime(['2001-03-31 00:00:00',
                       '2002-05-31 00:00:00',
                       '2003-08-31 00:00:00'],
                       dtype='datetime64[ns]')
```

```python
>>> idx
DatetimeIndex(['2001-03-31', '2002-05-31', '2003-08-31'],
              dtype='datetime64[ns]', freq=None)
```
```python
>>> idx.to_period("M")
PeriodIndex(['2001-03', '2002-05', '2003-08'], dtype='period[M]')
```

For the yearly frequency

```python
>>> idx.to_period("Y")
PeriodIndex(['2001', '2002', '2003'], dtype='period[A-DEC]')
```

**pandas.DataFrame.to_records**

DataFrame.to_records(index=True, column_dtypes=None, index_dtypes=None)

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

**Parameters**

- **index** [bool, default True] Include index in resulting record array, stored in ‘index’ field or using the index label, if set.
- **column_dtypes** [str, type, dict, default None] If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.
- **index_dtypes** [str, type, dict, default None] If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.

This mapping is applied only if index=True.

**Returns**

- **numpy.recarray** NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

**See also:**

- **DataFrame.from_records** Convert structured or record ndarray to DataFrame.
- **numpy.recarray** An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

**Examples**

```python
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
                    index=['a', 'b'])
>>> df
   A  B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<f8'), ('B', '<f8')])
```

If the DataFrame index has no label then the recarray field name is set to ‘index’. If the index has a label then this is used as the field name.
>>> df.index = df.index.rename("I")
>>> df.to_records()
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],
dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
The index can be excluded from the record array:

>>> df.to_records(index=False)
rec.array([[1, 0.5], [2, 0.75]],
dtype=[('A', '<i8'), ('B', '<f8')])

Data types can be specified for the columns:

>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],
dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])

As well as for the index:

>>> index_dtypes = f"<S{df.index.str.len().max()}
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],
dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])

pandas.DataFrame.to_string

DataFrame.to_string(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, max_rows=None, max_cols=None, show_dimensions=False, decimal='.', line_width=None, min_rows=None, max_colwidth=None, encoding=None)

Render a DataFrame to a console-friendly tabular output.

Parameters

buf [str, Path or StringIO-like, optional, default None] Buffer to write to. If None, the output is returned as a string.

columns [sequence, optional, default None] The subset of columns to write. Writes all columns by default.

col_space [int, list or dict of int, optional] The minimum width of each column. If a list of ints is given every integers corresponds with one column. If a dict is given, the key references the column, while the value defines the space to use.

header [bool or sequence of str, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, optional, default True] Whether to print index (row) labels.

na_rep [str, optional, default 'NaN'] String representation of NaN to use.
formatters  [list, tuple or dict of one-param. functions, optional] Formatter functions to apply to columns’ elements by position or name. The result of each function must be a unicode string. List/tuple must be of length equal to the number of columns.

float_format  [one-parameter function, optional, default None] Formatter function to apply to columns’ elements if they are floats. This function must return a unicode string and will be applied only to the non-NaN elements, with NaN being handled by na_rep.

Changed in version 1.2.0.

sparsify  [bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names  [bool, optional, default True] Prints the names of the indexes.

justify  [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box. Valid values are

• left
• right
• center
• justify
• justify-all
• start
• end
• inherit
• match-parent
• initial
• unset.

max_rows  [int, optional] Maximum number of rows to display in the console.

max_cols  [int, optional] Maximum number of columns to display in the console.

show_dimensions  [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal  [str, default ‘.’] Character recognized as decimal separator, e.g. ‘,’ in Europe.

line_width  [int, optional] Width to wrap a line in characters.

min_rows  [int, optional] The number of rows to display in the console in a truncated repr (when number of rows is above max_rows).

max_colwidth  [int, optional] Max width to truncate each column in characters. By default, no limit.

New in version 1.0.0.

encoding  [str, default “utf-8”] Set character encoding.

New in version 1.0.

Returns

str or None  If buf is None, returns the result as a string. Otherwise returns None.
See also:

**to_html**  Convert DataFrame to HTML.

**Examples**

```python
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
col1  col2
0 1 4
1 2 5
2 3 6
```

**pandas.DataFrame.to_timestamp**

DataFrame.to_timestamp(freq=None, how='start', axis=0, copy=True)

Cast to DatetimeIndex of timestamps, at beginning of period.

**Parameters**

- **freq** [str, default frequency of PeriodIndex] Desired frequency.
- **how** [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.
- **axis** [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).
- **copy** [bool, default True] If False then underlying input data is not copied.

**Returns**

DataFrame with DatetimeIndex

**pandas.DataFrame.to_xarray**

DataFrame.to_xarray()

Return an xarray object from the pandas object.

**Returns**

xarray.DataArray or xarray.Dataset Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

- **DataFrame.to_hdf** Write DataFrame to an HDF5 file.
- **DataFrame.to_parquet** Write a DataFrame to the binary parquet format.
Notes

See the xarray docs

Examples

```python
>>> df = pd.DataFrame([(
'falcon', 'bird', 389.0, 2),
'parrot', 'bird', 24.0, 2),
'lion', 'mammal', 80.5, 4),
'monkey', 'mammal', np.nan, 4)],
columns=['name', 'class', 'max_speed',
'num_legs'])
>>> df
   name   class  max_speed  num_legs
0  falcon   bird     389.0       2
1  parrot   bird      24.0       2
2   lion  mammal     80.5       4
3  monkey  mammal      NaN       4
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions: (animal: 2, date: 2)
Coordinates:
  * date (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal (animal) object 'falcon' 'parrot'
Data variables:
  speed (date, animal) int64 350 18 361 15
```

**pandas.DataFrame.transform**

DataFrame.transform(func, axis=0, *args, **kwargs)

Call func on self producing a DataFrame with the same axis shape as self.

**Parameters**

- **func** [function, str, list-like or dict-like] Function to use for transforming the data.
  - If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.
  - Accepted combinations are:
    - function
    - string function name
    - list-like of functions and/or function names, e.g. [np.exp, 'sqrt']
    - dict-like of axis labels -> functions, function names or list-like of such.

- **axis** [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

- **args** Positional arguments to pass to func.

- **kwargs** Keyword arguments to pass to func.

**Returns**

DataFrame A DataFrame that must have the same length as self.

**Raises**

- **ValueError** [If the returned DataFrame has a different length than self.]

**See also:**

- **DataFrame.agg** Only perform aggregating type operations.
- **DataFrame.apply** Invoke function on a DataFrame.

3.4. DataFrame 1941
Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```python
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
```
```
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```python
>>> s = pd.Series(range(3))
>>> s
0  0
1  1
2  2
```
```
>>> s.transform([np.sqrt, np.exp])
   sqrt           exp
0  0.000000  1.000000
1  1.000000  2.718282
2  1.414214  7.389056
```

You can call transform on a GroupBy object:

```python
>>> df = pd.DataFrame({
...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
```
```
>>> df.groupby('Date')['Data'].transform('sum')
```
```
pandas.DataFrame.transpose

DataFrame.transpose(*args, copy=False)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property $T$ is an accessor to the method transpose().

**Parameters**

*args [tuple, optional] Accepted for compatibility with NumPy.

**copy** [bool, default False] Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

**Returns**
DataFrame  The transposed DataFrame.

See also:

numpy.transpose  Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```python
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
      col1  col2
0      1    3
1      2    4
```

```python
>>> df1_transposed = df1.T  # or df1.transpose()
```

```python
>>> df1_transposed
      0  1
   col1  1  2
   col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```python
>>> df1.dtypes
col1  int64
col2  int64
dtype: object
```

```python
>>> df1_transposed.dtypes
0  int64
1  int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```python
>>> d2 = {'name': ['Alice', 'Bob'],
       ...       'score': [9.5, 8],
       ...       'employed': [False, True],
       ...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
     name  score employed  kids
0    Alice   9.5    False      0
1      Bob   8.0     True      0
```
When the DataFrame has mixed dtypes, we get a transposed DataFrame with the object dtype:

```python
>>> df2.dtypes
name    object
score  float64
employed  bool
kids     int64
dtype: object
```

```python
>>> df2_transposed.dtypes
0      object
1      object
dtype: object
```

### pandas.DataFrame.truediv

**DataFrame.truediv**(other, axis='columns', level=None, fill_value=None)  

Get Floating division of dataframe and other, element-wise (binary operator truediv).  

Equivalent to dataframe / other, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, rtruediv.  

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.  

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.

**See also:**

- **DataFrame.add** Add DataFrames.
- **DataFrame.sub** Subtract DataFrames.
- **DataFrame.mul** Multiply DataFrames.
**DataFrame.div** Divide DataFrames (float division).

**DataFrame.truediv** Divide DataFrames (float division).

**DataFrame.floordiv** Divide DataFrames (integer division).

**DataFrame.mod** Calculate modulo (remainder after division).

**DataFrame.pow** Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],
        'degrees': [360, 180, 360],
        index=['circle', 'triangle', 'rectangle'])
>>> df
    angles  degrees
circle    0     360
triangle   3    180
rectangle  4     360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
     angles  degrees
circle     1     361
triangle    4    181
rectangle   5     361
```

```python
>>> df.add(1)
     angles  degrees
circle     1     361
triangle    4    181
rectangle   5     361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
     angles  degrees
circle   0.0    36.0
triangle 0.3    18.0
rectangle 0.4    36.0
```

```python
>>> df.rdiv(10)
     angles  degrees
circle  inf   0.027778
triangle 3.333333      0.055556
rectangle 2.500000   0.027778
```

Subtract a list and Series by axis with operator version.
Multiply a DataFrame of different shape with operator version.

```python
circle 0  NaN
triangle 9  NaN
rectangle 16 NaN
```

Divide by a MultiIndex by level.

```python
circle 0  0.0
triangle 9  0.0
rectangle 16 0.0
```
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720

>>> df.div(df_multindex, level=1, fill_value=0)

<table>
<thead>
<tr>
<th>angles</th>
<th>degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>A circle</td>
<td>NaN 1.0</td>
</tr>
<tr>
<td>triangle</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>rectangle</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>B square</td>
<td>0.0 0.0</td>
</tr>
<tr>
<td>pentagon</td>
<td>0.0 0.0</td>
</tr>
<tr>
<td>hexagon</td>
<td>0.0 0.0</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.truncate**

DataFrame.truncate(before=None, after=None, axis=None, copy=True)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

**Parameters**

- **before** [date, str, int] Truncate all rows before this index value.
- **after** [date, str, int] Truncate all rows after this index value.
- **axis** [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.
- **copy** [bool, default is True] Return a copy of the truncated section.

**Returns**

- **type of caller** The truncated Series or DataFrame.

**See also:**

- **DataFrame.loc** Select a subset of a DataFrame by label.
- **DataFrame.iloc** Select a subset of a DataFrame by position.

**Notes**

If the index being truncated contains only datetime values, before and after may be specified as strings instead of Timestamps.
Examples

```python
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
                    'B': ['f', 'g', 'h', 'i', 'j'],
                    'C': ['k', 'l', 'm', 'n', 'o']},
                   index=[1, 2, 3, 4, 5])
>>> df
   A   B   C
0  a  f  k
1  b  g  l
2  c  h  m
3  d  i  n
4  e  j  o

>>> df.truncate(before=2, after=4)
   A   B   C
0  b  g  l
1  c  h  m
2  d  i  n

The columns of a DataFrame can be truncated.

```python
>>> df.truncate(before="A", after="B", axis="columns")
   A   B
0  a  f
1  b  g
2  c  h
3  d  i
4  e  j
```

For Series, only rows can be truncated.

```python
>>> df['A'].truncate(before=2, after=4)
2  b
3  c
4  d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```python
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                  A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1

>>> df.truncate(before=pd.Timestamp('2016-01-05'),
               after=pd.Timestamp('2016-01-10')).tail()
                  A
```

(continues on next page)
Because the index is a DatetimeIndex containing only dates, we can specify `before` and `after` as strings. They will be coerced to Timestamps before truncation.

```python
>>> df.truncate('2016-01-05', '2016-01-10').tail()
   2016-01-09 23:59:56 1
   2016-01-09 23:59:57 1
   2016-01-09 23:59:58 1
   2016-01-09 23:59:59 1
   2016-01-10 00:00:00 1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```python
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
   2016-01-10 23:59:55 1
   2016-01-10 23:59:56 1
   2016-01-10 23:59:57 1
   2016-01-10 23:59:58 1
   2016-01-10 23:59:59 1
```

### pandas.DataFrame.tshift

**DataFrame.tshift**(periods=1, freq=None, axis=0)

Shift the time index, using the index’s frequency if available.

Deprecated since version 1.1.0: Use shift instead.

**Parameters**

- **periods** [int] Number of periods to move, can be positive or negative.
- **freq** [DateOffset, timedelta, or str, default None] Increment to use from the tseries module or time rule expressed as a string (e.g. ‘EOM’).
- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default 0] Corresponds to the axis that contains the Index.

**Returns**

- **shifted** [Series/DataFrame]
Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

pandas.DataFrame.tz_convert

DataFrame.tz_convert(tz, axis=0, level=None, copy=True)

Convert tz-aware axis to target time zone.

Parameters

tz  [str or tzinfo object]
axis [the axis to convert]
level [int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.
copy [bool, default True] Also make a copy of the underlying data.

Returns

{klass} Object with time zone converted axis.

Raises

TypeError If the axis is tz-naive.

pandas.DataFrame.tz_localize

DataFrame.tz_localize(tz, axis=0, level=None, copy=True, ambiguous='raise', nonexistent='raise')

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use Series.dt.tz_localize().

Parameters

tz  [str or tzinfo]
axis [the axis to localize]
level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None.
copy [bool, default True] Also make a copy of the underlying data.
ambiguous ['infer', bool-narray, ‘NaT’, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the ambiguous parameter dictates how ambiguous times should be handled.

• ‘infer’ will attempt to infer fall dst-transition hours based on order
• bool-narray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
• ‘NaT’ will return NaT where there are ambiguous times
• ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

**nonexistent** [str, default ‘raise’] A nonexistent time does not exist in a particular time-zone where clocks moved forward due to DST. Valid values are:

• ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
• ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
• ‘NaT’ will return NaT where there are nonexistent times
• timedelta objects will shift nonexistent times by the timedelta
• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

Series or DataFrame Same type as the input.

**Raises**

TypeError If the TimeSeries is tz-aware and tz is not None.

**Examples**

Localize local times:

```python
>>> s = pd.Series([1],
    ...     index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00 1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```python
>>> s = pd.Series(range(7),
    ...     index=pd.DatetimeIndex(['2018-10-28 01:30:00',
    ...         '2018-10-28 02:00:00',
    ...         '2018-10-28 02:30:00',
    ...         '2018-10-28 02:00:00',
    ...         '2018-10-28 02:30:00',
    ...         '2018-10-28 03:00:00',
    ...         '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00 0
2018-10-28 02:00:00+02:00 1
2018-10-28 02:30:00+02:00 2
2018-10-28 02:00:00+01:00 3
2018-10-28 02:30:00+01:00 4
2018-10-28 03:00:00+01:00 5
2018-10-28 03:30:00+01:00 6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly.
If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or 'shift_forward' or 'shift_backward'.

```python
>>> s = pd.Series(range(2),
    index=pd.DatetimeIndex(['2015-03-29 02:30:00',
                            '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
```

```
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
```

`s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))`

```
2015-03-29 03:30:00+02:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
```

---

**pandas.DataFrame.unstack**

DataFrame.unstack(level=-1, fill_value=None)

Pivot a level of the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

Parameters

- **level** [int, str, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name.

- **fill_value** [int, str or dict] Replace NaN with this value if the unstack produces missing values.

Returns

Series or DataFrame

See also:

DataFrame.pivot Pivot a table based on column values.
**DataFrame.stack**  Pivot a level of the column labels (inverse operation from `unstack`).

**Notes**

Reference *the user guide* for more examples.

**Examples**

```python
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
                                      ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one a 1.0
    b 2.0
two a 3.0
    b 4.0
dtype: float64
```

```python
>>> s.unstack(level=-1)
a  
  one 1.0 2.0
  two 3.0 4.0

>>> s.unstack(level=0)
   
  one two
    a 1.0 3.0
    b 2.0 4.0

>>> df = s.unstack(level=0)
>>> df
      
  one  
    a 1.0
    b 2.0
  two  
    a 3.0
    b 4.0
dtype: float64
```

**pandas.DataFrame.update**

DataFrame.update(*other*, join='left', overwrite=True, filter_func=None, errors='ignore')

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

**Parameters**

*other*  [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

*join*  [['left'], default 'left'] Only left join is implemented, keeping the index and columns of the original object.
overwrite [bool, default True] How to handle non-NA values for overlapping keys:
- True: overwrite original DataFrame’s values with values from other.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array)->bool1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors [‘raise’, ‘ignore’], default ‘ignore’] If ‘raise’, will raise a ValueError if the DataFrame and other both contain non-NA data in the same place.

Returns
None [method directly changes calling object]

Raises

ValueError
- When errors=’raise’ and there’s overlapping non-NA data.
- When errors is not either ‘ignore’ or ‘raise’

NotImplementedError
- If join != ‘left’

See also:
dict.update Similar method for dictionaries.
DataFrame.merge For column(s)-on-column(s) operations.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 3],
... 'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
... 'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
  A  B
0 1  4
1 2  5
2 3  6
```

The DataFrame’s length does not increase as a result of the update, only values at matching index/column labels are updated.

```python
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
... 'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
  A  B
0  a  d
1  b  e
2  c  f
```
For Series, its name attribute must be set.

```python
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
      A  B
0    a  d
1    b  y
2    c  e
```

If `other` contains NaNs the corresponding values are not updated in the original dataframe.

```python
>>> df = pd.DataFrame({'A': [1, 2, 3],
                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4.0
1  2 500.0
2  3  6.0
```

`pandas.DataFrame.value_counts`

DataFrame.value_counts(subset=None, normalize=False, sort=True, ascending=False, dropna=True)

Return a Series containing counts of unique rows in the DataFrame.

New in version 1.1.0.

Parameters

- **subset** [list-like, optional] Columns to use when counting unique combinations.
- **normalize** [bool, default False] Return proportions rather than frequencies.
- **sort** [bool, default True] Sort by frequencies.
- **ascending** [bool, default False] Sort in ascending order.
- **dropna** [bool, default True] Don't include counts of rows that contain NA values.

New in version 1.3.0.

Returns

Series
See also:

Series.value_counts  Equivalent method on Series.

Notes

The returned Series will have a MultiIndex with one level per input column. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```python
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0],
...                    index=['falcon', 'dog', 'cat', 'ant'])

>>> df
     num_legs  num_wings
falcon     2          2
dog        4          0
cat        4          0
ant        6          0

>>> df.value_counts()
     num_legs  num_wings
2          0          2
4          2          1
6          0          1
dtype: int64

>>> df.value_counts(sort=False)
     num_legs  num_wings
2          2          1
4          0          2
6          0          1
dtype: int64

>>> df.value_counts(ascending=True)
     num_legs  num_wings
2          2          1
6          0          1
4          0          2
dtype: int64

>>> df.value_counts(normalize=True)
     num_legs  num_wings
4          0          0.50
2          2          0.25
6          0          0.25
dtype: float64
```

With dropna set to False we can also count rows with NA values.
```python
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
                     'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
```
```
0  John     Smith
1  Anne      <NA>
2  John      <NA>
3  Beth  Louise
```
```
>>> df
first_name  middle_name
0   John        Smith
1  Anne           <NA>
2  John          <NA>
3   Beth    Louise
```
```
>>> df.value_counts()
first_name  middle_name
Beth   Louise      1
John   Smith      1
dtype: int64
```
```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne       NaN       1
Beth   Louise      1
John   Smith      1
          NaN       1
dtype: int64
```
```
pandas.DataFrame.var

DataFrame.var(axis=None, skipna=True, level=None, ddof=1, numeric_only=None, **kwargs)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

axis  [[index (0), columns (1)]]

skipna  [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level  [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

ddof  [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only  [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

Series or DataFrame (if level specified)
Examples

```python
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]})
...                .set_index('person_id')

>>> df
   person_id  age  height
0          0   21   1.61
1          1   25   1.87
2          2   62   1.49
3          3   43   2.01

>>> df.var()
   age   height
0  352.92  0.05637

Alternatively, ddof=0 can be set to normalize by N instead of N-1:

>>> df.var(ddof=0)
   age   height
0  264.69  0.04228
```

pandas.DataFrame.where

DataFrame.where(cond, other=NoDefault.no_default, inplace=False, axis=None, errors='raise', try_cast=NoDefault.no_default)

Replace values where the condition is False.

Parameters

- **cond** [bool Series/DataFrame, array-like, or callable] Where `cond` is True, keep the original value. Where False, replace with corresponding value from `other`. If `cond` is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

- **other** [scalar, Series/DataFrame, or callable] Entries where `cond` is False are replaced with corresponding value from `other`. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn’t check it).

- **inplace** [bool, default False] Whether to perform the operation in place on the data.

- **axis** [int, default None] Alignment axis if needed.

- **level** [int, default None] Alignment level if needed.

- **errors** [str, {'raise', 'ignore'}, default 'raise'] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.
  - 'raise': allow exceptions to be raised.
  - 'ignore': suppress exceptions. On error return original object.
try_cast  [bool, default None] Try to cast the result back to the input type (if possible).

Deprecated since version 1.3.0: Manually cast back if necessary.

Returns

Same type as caller or None if inplace=True.

See also:

DataFrame.mask()  Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if cond is True the element is used; otherwise the corresponding element from the DataFrame other is used.

The signature for DataFrame.where() differs from numpy.where(). Roughly df1.where(m, df2) is equivalent to np.where(m, df1, df2).

For further details and examples see the where documentation in indexing.

Examples

```python
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```python
>>> s.where(s > 1, 10)
0     10
1     10
2     2
3     3
4     4
dtype: int64
>>> s.mask(s > 1, 10)
0     0
1     1
2     10
3     10
4     10
dtype: int64
```
```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3 -6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

**pandas.DataFrame.xs**

DataFrame.xs(key, axis=0, level=None, drop_level=True)

Return cross-section from the Series/DataFrame.

This method takes a key argument to select data at a particular level of a MultiIndex.

**Parameters**

- **key** [label or tuple of label] Label contained in the index, or partially in a MultiIndex.
- **axis** [[0 or ‘index’, 1 or ‘columns’], default 0] Axis to retrieve cross-section on.
- **level** [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.
- **drop_level** [bool, default True] If False, returns object with same levels as self.

**Returns**

- **Series or DataFrame** Cross-section from the original Series or DataFrame corresponding to the selected index levels.

**See also:**
**DataFrame.loc**  Access a group of rows and columns by label(s) or a boolean array.

**DataFrame.iloc**  Purely integer-location based indexing for selection by position.

**Notes**

`xs` cannot be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of `xs` functionality, see MultiIndex Slicers.

**Examples**

```python
>>> d = {
...     'num_legs': [4, 4, 2, 2],
...     'num_wings': [0, 0, 2, 2],
...     'class': ['mammal', 'mammal', 'mammal', 'bird'],
...     'animal': ['cat', 'dog', 'bat', 'penguin'],
...     'locomotion': ['walks', 'walks', 'flies', 'walks']
... }
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df

<table>
<thead>
<tr>
<th>class</th>
<th>animal</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>mammal</td>
<td>cat</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dog</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bat</td>
<td>flies</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>bird</td>
<td>penguin</td>
<td>walks</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Get values at specified index

```python
>>> df.xs('mammal')
```

<table>
<thead>
<tr>
<th>animal</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>dog</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>bat</td>
<td>flies</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Get values at several indexes

```python
>>> df.xs(('mammal', 'dog'))
```

<table>
<thead>
<tr>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Get values at specified index and level

```python
>>> df.xs('cat', level=1)
```

<table>
<thead>
<tr>
<th>class</th>
<th>locomotion</th>
<th>num_legs</th>
<th>num_wings</th>
</tr>
</thead>
<tbody>
<tr>
<td>mammal</td>
<td>walks</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Get values at several indexes and levels
### 3.4.2 Attributes and underlying data

**Axes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.index</code></td>
<td>The index (row labels) of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.columns</code></td>
<td>The column labels of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.dtypes</code></td>
<td>Return the dtypes in the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.info</code></td>
<td>Print a concise summary of a DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.select_dtypes</code></td>
<td>Return a subset of the DataFrame's columns based on the column dtypes.</td>
</tr>
<tr>
<td><code>DataFrame.values</code></td>
<td>Return a Numpy representation of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.axes</code></td>
<td>Return a list representing the axes of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.ndim</code></td>
<td>Return an int representing the number of axes / array dimensions.</td>
</tr>
<tr>
<td><code>DataFrame.size</code></td>
<td>Return an int representing the number of elements in this object.</td>
</tr>
<tr>
<td><code>DataFrame.shape</code></td>
<td>Return a tuple representing the dimensionality of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.memory_usage</code></td>
<td>Return the memory usage of each column in bytes.</td>
</tr>
<tr>
<td><code>DataFrame.empty</code></td>
<td>Indicator whether Series/DataFrame is empty.</td>
</tr>
<tr>
<td><code>DataFrame.set_flags</code></td>
<td>Return a new object with updated flags.</td>
</tr>
</tbody>
</table>

### 3.4.3 Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.astype</code></td>
<td>Cast a pandas object to a specified dtype dtype.</td>
</tr>
<tr>
<td><code>DataFrame.convert_dtypes</code></td>
<td>Convert columns to best possible dtypes using dtypes supporting pd.NA.</td>
</tr>
<tr>
<td><code>DataFrame.infer_objects</code></td>
<td>Attempt to infer better dtypes for object columns.</td>
</tr>
<tr>
<td><code>DataFrame.copy</code></td>
<td>Make a copy of this object's indices and data.</td>
</tr>
<tr>
<td><code>DataFrame.bool</code></td>
<td>Return the bool of a single element Series or DataFrame.</td>
</tr>
</tbody>
</table>

3.4. DataFrame 1963
3.4.4 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.head</td>
<td>Return the first ( n ) rows.</td>
</tr>
<tr>
<td>DataFrame.at</td>
<td>Access a single value for a row/column label pair.</td>
</tr>
<tr>
<td>DataFrame.iat</td>
<td>Access a single value for a row/column pair by integer position.</td>
</tr>
<tr>
<td>DataFrame.loc</td>
<td>Access a group of rows and columns by label(s) or a boolean array.</td>
</tr>
<tr>
<td>DataFrame.iloc</td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td>DataFrame.insert</td>
<td>Insert column into DataFrame at specified location.</td>
</tr>
<tr>
<td>DataFrame.items</td>
<td>Iterate over info axis.</td>
</tr>
<tr>
<td>DataFrame.iteritems</td>
<td>Iterate over (column name, Series) pairs.</td>
</tr>
<tr>
<td>DataFrame.keys</td>
<td>Get the 'info axis' (see Indexing for more).</td>
</tr>
<tr>
<td>DataFrame.iterrows</td>
<td>Iterate over DataFrame rows as (index, Series) pairs.</td>
</tr>
<tr>
<td>DataFrame.itertuples</td>
<td>Iterate over DataFrame rows as namedtuples.</td>
</tr>
<tr>
<td>DataFrame.lookup</td>
<td>(DEPRECATED) Label-based &quot;fancy indexing&quot; function for DataFrame.</td>
</tr>
<tr>
<td>DataFrame.pop</td>
<td>Return item and drop from frame.</td>
</tr>
<tr>
<td>DataFrame.tail</td>
<td>Return the last ( n ) rows.</td>
</tr>
<tr>
<td>DataFrame.xs</td>
<td>Return cross-section from the Series/DataFrame.</td>
</tr>
<tr>
<td>DataFrame.get</td>
<td>Get item from object for given key (ex: DataFrame column).</td>
</tr>
<tr>
<td>DataFrame.isin</td>
<td>Whether each element in the DataFrame is contained in values.</td>
</tr>
<tr>
<td>DataFrame.where</td>
<td>Replace values where the condition is False.</td>
</tr>
<tr>
<td>DataFrame.mask</td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td>DataFrame.query</td>
<td>Query the columns of a DataFrame with a boolean expression.</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.\_\_iter\_\_**

DataFrame.\_\_iter\_\_()

Iterate over info axis.

**Returns**

iterator  Info axis as iterator.

For more information on .at, .iat, .loc, and .iloc, see the *indexing documentation*. 

---

1964  Chapter 3.  API reference
### 3.4.5 Binary operator functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.add(other[, axis, level, fill_value])</code></td>
<td>Get Addition of dataframe and other, element-wise (binary operator <code>add</code>).</td>
</tr>
<tr>
<td><code>DataFrame.sub(other[, axis, level, fill_value])</code></td>
<td>Get Subtraction of dataframe and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.mul(other[, axis, level, fill_value])</code></td>
<td>Get Multiplication of dataframe and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.div(other[, axis, level, fill_value])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>DataFrame.truediv(other[, axis, level, ...])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>true_div</code>).</td>
</tr>
<tr>
<td><code>DataFrame.floordiv(other[, axis, level, ...])</code></td>
<td>Get Integer division of dataframe and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.mod(other[, axis, level, fill_value])</code></td>
<td>Get Modulo of dataframe and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.pow(other[, axis, level, fill_value])</code></td>
<td>Get Exponential power of dataframe and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.dot(other)</code></td>
<td>Compute the matrix multiplication between the DataFrame and other.</td>
</tr>
<tr>
<td><code>DataFrame.radd(other[, axis, level, fill_value])</code></td>
<td>Get Addition of dataframe and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rsub(other[, axis, level, fill_value])</code></td>
<td>Get Subtraction of dataframe and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmul(other[, axis, level, fill_value])</code></td>
<td>Get Multiplication of dataframe and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rdiv(other[, axis, level, fill_value])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>rtrue_div</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rtruediv(other[, axis, level, ...])</code></td>
<td>Get Floating division of dataframe and other, element-wise (binary operator <code>rtrue_div</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rfloordiv(other[, axis, level, ...])</code></td>
<td>Get Integer division of dataframe and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmod(other[, axis, level, fill_value])</code></td>
<td>Get Modulo of dataframe and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rpow(other[, axis, level, fill_value])</code></td>
<td>Get Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.lt(other[, axis, level])</code></td>
<td>Get Less than of dataframe and other, element-wise (binary operator <code>lt</code>).</td>
</tr>
<tr>
<td><code>DataFrame.gt(other[, axis, level])</code></td>
<td>Get Greater than of dataframe and other, element-wise (binary operator <code>gt</code>).</td>
</tr>
<tr>
<td><code>DataFrame.le(other[, axis, level])</code></td>
<td>Get Less than or equal to of dataframe and other, element-wise (binary operator <code>le</code>).</td>
</tr>
<tr>
<td><code>DataFrame.ge(other[, axis, level])</code></td>
<td>Get Greater than or equal to of dataframe and other, element-wise (binary operator <code>ge</code>).</td>
</tr>
<tr>
<td><code>DataFrame.ne(other[, axis, level])</code></td>
<td>Get Not equal to of dataframe and other, element-wise (binary operator <code>ne</code>).</td>
</tr>
<tr>
<td><code>DataFrame.eq(other[, axis, level])</code></td>
<td>Get Equal to of dataframe and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
<tr>
<td><code>DataFrame.combine(other, func[, fill_value, ...])</code></td>
<td>Perform column-wise combine with another DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.combine_first(other)</code></td>
<td>Update null elements with value in the same location in other.</td>
</tr>
</tbody>
</table>
3.4.6 Function application, GroupBy & window

- `DataFrame.apply(func[, axis, raw, ...])`: Apply a function along an axis of the DataFrame.
- `DataFrame.applymap(func[, na_action])`: Apply a function to a Dataframe elementwise.
- `DataFrame.pipe(func, *args, **kwargs)`: Apply chainable functions that expect Series or DataFrames.
- `DataFrame.agg([func, axis])`: Aggregate using one or more operations over the specified axis.
- `DataFrame.aggregate([func, axis])`: Aggregate using one or more operations over the specified axis.
- `DataFrame.transform(func[, axis])`: Call func on self producing a DataFrame with the same axis shape as self.
- `DataFrame.groupby([by, axis, level, ...])`: Group DataFrame using a mapper or by a Series of columns.
- `DataFrame.rolling(window[, min_periods, ...])`: Provide rolling window calculations.
- `DataFrame.expanding([min_periods, center, ...])`: Provide expanding window calculations.
- `DataFrame.ewm([com, span, halflife, alpha, ...])`: Provide exponentially weighted (EW) calculations.

3.4.7 Computations / descriptive stats

- `DataFrame.abs()`: Return a Series/DataFrame with absolute numeric value of each element.
- `DataFrame.all([axis, bool_only, skipna, level])`: Return whether all elements are True, potentially over an axis.
- `DataFrame.any([axis, bool_only, skipna, level])`: Return whether any element is True, potentially over an axis.
- `DataFrame.clip([lower, upper, axis, inplace])`: Trim values at input threshold(s).
- `DataFrame.corr([method, min_periods])`: Compute pairwise correlation of columns, excluding NA/null values.
- `DataFrame.corrwith(other[, axis, drop, method])`: Compute pairwise correlation.
- `DataFrame.count([axis, level, numeric_only])`: Count non-NA cells for each column or row.
- `DataFrame.cov([min_periods, ddof])`: Compute pairwise covariance of columns, excluding NA/null values.
- `DataFrame.cummax([axis, skipna])`: Return cumulative maximum over a DataFrame or Series axis.
- `DataFrame.cummin([axis, skipna])`: Return cumulative minimum over a DataFrame or Series axis.
- `DataFrame.cumprod([axis, skipna])`: Return cumulative product over a DataFrame or Series axis.
- `DataFrame.cumsum([axis, skipna])`: Return cumulative sum over a DataFrame or Series axis.
- `DataFrame.describe([percentiles, include, ...])`: Generate descriptive statistics.
- `DataFrame.diff([periods, axis])`: First discrete difference of element.
- `DataFrame.eval(expr[, inplace])`: Evaluate a string describing operations on DataFrame columns.
- `DataFrame.kurt([axis, skipna, level, ...])`: Return unbiased kurtosis over requested axis.
- `DataFrame.kurtosis([axis, skipna, level, ...])`: Return unbiased kurtosis over requested axis.
- `DataFrame.mad([axis, skipna, level])`: Return the mean absolute deviation of the values over the requested axis.

continues on next page
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.max</td>
<td>Return the maximum of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.mean</td>
<td>Return the mean of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.median</td>
<td>Return the median of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.min</td>
<td>Return the minimum of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.mode</td>
<td>Get the mode(s) of each element along the selected axis.</td>
</tr>
<tr>
<td>DataFrame.pct_change</td>
<td>Percentage change between the current and a prior element.</td>
</tr>
<tr>
<td>DataFrame.prod</td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.product</td>
<td>Return the product of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.quantile</td>
<td>Return values at the given quantile over requested axis.</td>
</tr>
<tr>
<td>DataFrame.rank</td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td>DataFrame.round</td>
<td>Round a DataFrame to a variable number of decimal places.</td>
</tr>
<tr>
<td>DataFrame.sem</td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td>DataFrame.skew</td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td>DataFrame.sum</td>
<td>Return the sum of the values over the requested axis.</td>
</tr>
<tr>
<td>DataFrame.std</td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td>DataFrame.var</td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td>DataFrame.nunique</td>
<td>Count number of distinct elements in specified axis.</td>
</tr>
<tr>
<td>DataFrame.value_counts</td>
<td>Return a Series containing counts of unique rows in the DataFrame.</td>
</tr>
</tbody>
</table>

### 3.4. DataFrame
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.add_prefix(prefix)</td>
<td>Prefix labels with string <code>prefix</code>.</td>
</tr>
<tr>
<td>DataFrame.add_suffix(suffix)</td>
<td>Suffix labels with string <code>suffix</code>.</td>
</tr>
<tr>
<td>DataFrame.align(other[, join, axis, level,...])</td>
<td>Align two objects on their axes with the specified join method.</td>
</tr>
<tr>
<td>DataFrame.at_time(time[, asof, axis])</td>
<td>Select values at particular time of day (e.g., 9:30AM).</td>
</tr>
<tr>
<td>DataFrame.between_time(start_time, end_time)</td>
<td>Select values between particular times of the day (e.g., 9:00-9:30 AM).</td>
</tr>
<tr>
<td>DataFrame.drop([labels, axis, index,...])</td>
<td>Drop specified labels from rows or columns.</td>
</tr>
<tr>
<td>DataFrame.drop_duplicates([subset, keep,...])</td>
<td>Return DataFrame with duplicate rows removed.</td>
</tr>
<tr>
<td>DataFrame.duplicated([subset, keep])</td>
<td>Return boolean Series denoting duplicate rows.</td>
</tr>
<tr>
<td>DataFrame.equals(other)</td>
<td>Test whether two objects contain the same elements.</td>
</tr>
<tr>
<td>DataFrame.filter([items, like, regex, axis])</td>
<td>Subset the dataframe rows or columns according to the specified index labels.</td>
</tr>
<tr>
<td>DataFrame.first(offset)</td>
<td>Select initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>DataFrame.head([n])</td>
<td>Return the first <code>n</code> rows.</td>
</tr>
<tr>
<td>DataFrame.idxmax([axis, skipna])</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td>DataFrame.idxmin([axis, skipna])</td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td>DataFrame.last(offset)</td>
<td>Select final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td>DataFrame.reindex([labels, index, columns,...])</td>
<td>Conform Series/DataFrame to new index with optional filling logic.</td>
</tr>
<tr>
<td>DataFrame.reindex_like(other[, method,...])</td>
<td>Return an object with matching indices as other object.</td>
</tr>
<tr>
<td>DataFrame.rename([mapper, index, columns,...])</td>
<td>Alter axes labels.</td>
</tr>
<tr>
<td>DataFrame.rename_axis([mapper, index,...])</td>
<td>Set the name of the axis for the index or columns.</td>
</tr>
<tr>
<td>DataFrame.reset_index([mapper, index,...])</td>
<td>Reset the index, or a level of it.</td>
</tr>
<tr>
<td>DataFrame.sample([n, frac, replace,...])</td>
<td>Return a random sample of items from an axis of object.</td>
</tr>
<tr>
<td>DataFrame.set_axis(labels[, axis, inplace])</td>
<td>Assign desired index to given axis.</td>
</tr>
<tr>
<td>DataFrame.set_index(keys[, drop, append,...])</td>
<td>Set the DataFrame index using existing columns.</td>
</tr>
<tr>
<td>DataFrame.tail([n])</td>
<td>Return the last <code>n</code> rows.</td>
</tr>
<tr>
<td>DataFrame.take([indices[, axis, is_copy]])</td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td>DataFrame.truncate([before, after, axis, copy])</td>
<td>Truncate a Series or DataFrame before and after some index value.</td>
</tr>
</tbody>
</table>
### 3.4.9 Missing data handling

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.backfill()</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.bfill()</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.dropna()</code></td>
<td>Remove missing values.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.ffill()</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='ffill'.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.fillna()</code></td>
<td>Fill NA/NaN values using the specified method.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.interpolate()</code></td>
<td>Fill NaN values using an interpolation method.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.isna()</code></td>
<td>Detect missing values.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.isnull()</code></td>
<td>DataFrame.isnull is an alias for DataFrame.isna.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.notna()</code></td>
<td>Detect existing (non-missing) values.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.notnull()</code></td>
<td>DataFrame.notnull is an alias for DataFrame.notna.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.pad()</code></td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='ffill'.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.replace()</code></td>
<td>Replace values given in <code>to_replace</code> with <code>value</code>.</td>
<td></td>
</tr>
</tbody>
</table>

### 3.4.10 Reshaping, sorting, transposing

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.droplevel()</code></td>
<td>Return Series/DataFrame with requested index / column level(s) removed.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.pivot()</code></td>
<td>Return reshaped DataFrame organized by given index / column values.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.pivot_table()</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.reorder_levels()</code></td>
<td>Rearrange index levels using input order.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.sort_values()</code></td>
<td>Sort by the values along either axis.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.sort_index()</code></td>
<td>Sort object by labels (along an axis).</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.nlargest()</code></td>
<td>Return the first n rows ordered by columns in descending order.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.nsmallest()</code></td>
<td>Return the first n rows ordered by columns in ascending order.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.swaplevel()</code></td>
<td>Swap levels i and j in a MultiIndex.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.stack()</code></td>
<td>Stack the prescribed level(s) from columns to index.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.unstack()</code></td>
<td>Pivot a level of the (necessarily hierarchical) index labels.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.swapaxes()</code></td>
<td>Interchange axes and swap values axes appropriately.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.melt()</code></td>
<td>Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.explode()</code></td>
<td>Squeeze 1 dimensional axis objects into scalars.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.to_xarray()</code></td>
<td>Return an xarray object from the pandas object.</td>
<td></td>
</tr>
<tr>
<td><code>DataFrame.T</code></td>
<td>Transpose index and columns.</td>
<td></td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DataFrame.T**

**property** DataFrame.T

### 3.4.11 Combining / comparing / joining / merging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.append(other[, ignore_index, ...])</td>
<td>(DEPRECATED) Append rows of other to the end of caller, returning a new object.</td>
</tr>
<tr>
<td>DataFrame.assign(**kwargs)</td>
<td>Assign new columns to a DataFrame.</td>
</tr>
<tr>
<td>DataFrame.compare(other[, align_axis, ...])</td>
<td>Compare to another DataFrame and show the differences.</td>
</tr>
<tr>
<td>DataFrame.join(other[, on, how, lsuffix, ...])</td>
<td>Join columns of another DataFrame.</td>
</tr>
<tr>
<td>DataFrame.merge(right[, how, on, left_on, ...])</td>
<td>Merge DataFrame or named Series objects with a database-style join.</td>
</tr>
<tr>
<td>DataFrame.update(other[, join, overwrite, ...])</td>
<td>Modify in place using non-NA values from another DataFrame.</td>
</tr>
</tbody>
</table>

### 3.4.12 Time Series-related

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.asfreq(freq[, method, how, ...])</td>
<td>Convert time series to specified frequency.</td>
</tr>
<tr>
<td>DataFrame.asof(where[, subset])</td>
<td>Return the last row(s) without any NaNs before where.</td>
</tr>
<tr>
<td>DataFrame.shift([periods, freq, axis, ...])</td>
<td>Shift index by desired number of periods with an optional time freq.</td>
</tr>
<tr>
<td>DataFrame.slice_shift([periods, axis])</td>
<td>(DEPRECATED) Equivalent to shift without copying data.</td>
</tr>
<tr>
<td>DataFrame.tshift([periods, freq, axis])</td>
<td>(DEPRECATED) Shift the time index, using the index's frequency if available.</td>
</tr>
<tr>
<td>DataFrame.first_valid_index()</td>
<td>Return index for first non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td>DataFrame.last_valid_index()</td>
<td>Return index for last non-NA value or None, if no non-NA value is found.</td>
</tr>
<tr>
<td>DataFrame.resample(rule[, axis, closed, ...])</td>
<td>Resample time-series data.</td>
</tr>
<tr>
<td>DataFrame.to_period([freq, axis, copy])</td>
<td>Convert DataFrame from DatetimeIndex to PeriodIndex.</td>
</tr>
<tr>
<td>DataFrame.to_timestamp([freq, how, axis, copy])</td>
<td>Cast to DatetimeIndex of timestamps, at beginning of period.</td>
</tr>
<tr>
<td>DataFrame.tz_convert(tz[, axis, level, copy])</td>
<td>Convert tz-aware axis to target time zone.</td>
</tr>
<tr>
<td>DataFrame.tz_localize(tz[, axis, level, ...])</td>
<td>Localize tz-naive index of a Series or DataFrame to target time zone.</td>
</tr>
</tbody>
</table>

### 3.4.13 Flags

Flags refer to attributes of the pandas object. Properties of the dataset (like the date it was recorded, the URL it was accessed from, etc.) should be stored in DataFrame.attrs.

**Flags(obj, *, allows_duplicate_labels)** Flags that apply to pandas objects.
3.4.14 Metadata

`DataFrame.attrs` is a dictionary for storing global metadata for this DataFrame.

**Warning:** `DataFrame.attrs` is considered experimental and may change without warning.

`DataFrame.attrs` Dictionary of global attributes of this dataset.

3.4.15 Plotting

`DataFrame.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `DataFrame.plot.<kind>`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.plot.area(x, y)</td>
<td>Draw a stacked area plot.</td>
</tr>
<tr>
<td>DataFrame.plot.bar(x, y)</td>
<td>Vertical bar plot.</td>
</tr>
<tr>
<td>DataFrame.plot.barh(x, y)</td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td>DataFrame.plot.box(by)</td>
<td>Make a box plot of the DataFrame columns.</td>
</tr>
<tr>
<td>DataFrame.plot.density(bw_method, ind)</td>
<td>Generate Kernel Density Estimate plot using Gaussian kernels.</td>
</tr>
<tr>
<td>DataFrame.plot.hexbin(x, y, C, ...)</td>
<td>Generate a hexagonal binning plot.</td>
</tr>
<tr>
<td>DataFrame.plot.hist(by, bins)</td>
<td>Draw one histogram of the DataFrame's columns.</td>
</tr>
<tr>
<td>DataFrame.plot.kde(bw_method, ind)</td>
<td>Generate Kernel Density Estimate plot using Gaussian kernels.</td>
</tr>
<tr>
<td>DataFrame.plot.line(x, y)</td>
<td>Plot Series or DataFrame as lines.</td>
</tr>
<tr>
<td>DataFrame.plot.pie(**kwargs)</td>
<td>Generate a pie plot.</td>
</tr>
<tr>
<td>DataFrame.plot.scatter(x, y, s, c)</td>
<td>Create a scatter plot with varying marker point size and color.</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.plot.area**

`DataFrame.plot.area(x=None, y=None, **kwargs)`  
Draw a stacked area plot.  
An area plot displays quantitative data visually. This function wraps the matplotlib area function.  

**Parameters**  
- `x` [label or position, optional] Coordinates for the X axis. By default uses the index.  
- `y` [label or position, optional] Column to plot. By default uses all columns.  
- `stacked` [bool, default True] Area plots are stacked by default. Set to False to create a un-stacked plot.  
- `**kwargs` Additional keyword arguments are documented in `DataFrame.plot()`.

**Returns**  
- `matplotlib.axes.Axes` or `numpy.ndarray` Area plot, or array of area plots if subplots is True.

**See also:**
**DataFrame.plot** Make plots of DataFrame using matplotlib / pylab.

**Examples**

Draw an area plot based on basic business metrics:

```python
>>> df = pd.DataFrame(
...     {'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
...     }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...     freq='M'))
>>> ax = df.plot.area()
```

Area plots are stacked by default. To produce an unstacked plot, pass `stacked=False`:

```python
>>> ax = df.plot.area(stacked=False)
```

Draw an area plot for a single column:

```python
>>> ax = df.plot.area(y='sales')
```

Draw with a different `x`:
```python
>>> df = pd.DataFrame(
...     {'sales': [3, 2, 3],
...      'visits': [20, 42, 28],
...      'day': [1, 2, 3],
...     })
>>> ax = df.plot.area(x='day')
```

```
pandas.DataFrame.plot.bar

DataFrame.plot.bar(x=None, y=None, **kwargs)

Vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

- `x` [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.
- `y` [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

pandas.DataFrame.plot.bar
```
**color**  [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:

- **A single color string referred to by name, RGB or RGBA code**, for instance ‘red’ or ‘#98d19’.
- **A sequence of color strings referred to by name, RGB or RGBA code**, which will be used for each column recursively. For instance ['green', 'yellow'] each column’s bar will be filled in green or yellow, alternatively. If there is only a single column to be plotted, then only the first color from the color list will be used.
- **A dict of the form {column name: color}**, so that each column will be colored accordingly. For example, if your columns are called a and b, then passing {'a': 'green', 'b': 'red'} will color bars for column a in green and bars for column b in red.

New in version 1.1.0.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

* [matplotlib.axes.Axes or np.ndarray of them](https://matplotlib.org/stable/api/axes_api.html) An ndarray is returned with one matplotlib.axes.Axes per column when subplots=True.

See also:

* DataFrame.plot.barh Horizontal bar plot.
* DataFrame.plot Make plots of a DataFrame.
* matplotlib.pyplot.bar Make a bar plot with matplotlib.

**Examples**

Basic plot.

```python
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> ax = df.plot.bar(x='lab', y='val', rot=0)
```

Plot a whole dataframe to a bar plot. Each column is assigned a distinct color, and each row is nested in a group along the horizontal axis.

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant', ... 'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed, ... 'lifespan': lifespan}, index=index)
>>> ax = df.plot.bar(rot=0)
```

Plot stacked bar charts for the DataFrame

```python
>>> ax = df.plot.bar(stacked=True)
```

Instead of nesting, the figure can be split by column with subplots=True. In this case, a numpy.ndarray of matplotlib.axes.Axes are returned.

```python
>>> axes = df.plot.bar(rot=0, subplots=True)
>>> axes[1].legend(loc=2)
```
If you don’t like the default colours, you can specify how you’d like each column to be colored.

```python
>>> axes = df.plot.bar(
    ...     rot=0, subplots=True, color={"speed": "red", "lifespan": "green"}
    ... )
>>> axes[1].legend(loc=2)
```

Plot a single column.

```python
>>> ax = df.plot.bar(y='speed', rot=0)
```

Plot only selected categories for the DataFrame.

```python
>>> ax = df.plot.bar(x='lifespan', rot=0)
```
DataFrame.plot.barh(x=None, y=None, **kwargs)

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to
the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot
shows the specific categories being compared, and the other axis represents a measured value.

Parameters

- **x** [label or position, optional] Allows plotting of one column versus another. If not specified,
  the index of the DataFrame is used.

- **y** [label or position, optional] Allows plotting of one column versus another. If not specified,
  all numerical columns are used.

- **color** [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:
  - A single color string referred to by name, RGB or RGBA code, for instance ‘red’ or ‘#a98d19’.
  - A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each column recursively. For instance ['green','yellow'] each column’s bar will be filled in green or yellow, alternatively. If there is only a single column to be plotted, then only the first color from the color list will be used.
  - A dict of the form {column name : color}, so that each column will be colored accordingly. For example, if your columns are called a and b, then passing {'a': 'green', 'b': 'red'} will color bars for column a in green and bars for column b in red.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

- matplotlib.axes.Axes or np.ndarray of them An ndarray is returned with one
  matplotlib.axes.Axes per column when subplots=True.

See also:

- DataFrame.plot.bar Vertical bar plot.
- DataFrame.plot Make plots of DataFrame using matplotlib.
- matplotlib.axes.Axes.bar Plot a vertical bar plot using matplotlib.

Examples

Basic example

```python
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> ax = df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
```
3.4. DataFrame
pandas: powerful Python data analysis toolkit, Release 1.4.2

... 'rabbit', 'giraffe', 'coyote', 'horse']
>> df = pd.DataFrame({'speed': speed,
... 'lifespan': lifespan}, index=index)
>> ax = df.plot.barh()

Plot stacked barh charts for the DataFrame

```python
>>> ax = df.plot.barh(stacked=True)
```

We can specify colors for each column

```python
>>> ax = df.plot.barh(color={'speed': "red", "lifespan": "green"})
```

Plot a column of the DataFrame to a horizontal bar plot

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
... 'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
... 'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(y='speed')
```
Plot DataFrame versus the desired column

```python
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
          ...   'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({
          ...   'speed': speed,
          ...   'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(x='lifespan')
```

**pandas.DataFrame.plot.box**

`DataFrame.plot.box(by=None, **kwargs)`

Make a box plot of the DataFrame columns.

A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to 1.5*IQR (IQR = Q3 - Q1) from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia’s entry for boxplot.

A consideration when using this chart is that the box and the whiskers can overlap, which is very common when plotting small sets of data.
Parameters

- **by** [str or sequence] Column in the DataFrame to group by.
  
  Changed in version 1.4.0: Previously, `by` is silently ignored and makes no groupings

- **kwargs** Additional keywords are documented in `DataFrame.plot()`.

Returns

- `matplotlib.axes.Axes` or `numpy.ndarray` of them

See also:

- `DataFrame.boxplot` Another method to draw a box plot.
- `Series.plot.box` Draw a box plot from a Series object.
- `matplotlib.pyplot.boxplot` Draw a box plot in matplotlib.

Examples

Draw a box plot from a DataFrame with four columns of randomly generated data.

```python
>>> data = np.random.randn(25, 4)
>>> df = pd.DataFrame(data, columns=list('ABCD'))
>>> ax = df.plot.box()
```

![Box plot example](image-url)
You can also generate groupings if you specify the `by` parameter (which can take a column name, or a list or tuple of column names):

Changed in version 1.4.0.

```python
>>> age_list = [8, 10, 12, 14, 72, 74, 76, 78, 20, 25, 30, 35, 60, 85]
>>> df = pd.DataFrame({"gender": list("MMMMMMMMFFFFFF"), "age": age_list})
>>> ax = df.plot.box(column="age", by="gender", figsize=(10, 8))
```

**pandas.DataFrame.plot.density**

`DataFrame.plot.density(bw_method=None, ind=None, **kwargs)`

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters**

- `bw_method` [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be ‘scott’, ‘silverman’, a scalar constant or a callable. If None (default), ‘scott’ is used. See `scipy.stats.gaussian_kde` for more information.
ind [NumPy array or int, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If ind is a NumPy array, the KDE is evaluated at the points passed. If ind is an integer, ind number of equally spaced points are used.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or numpy.ndarray of them

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```python
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:
Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

For DataFrame, it works in the same way:

```python
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```python
>>> ax = df.plot.kde(bw_method=0.3)

>>> ax = df.plot.kde(bw_method=3)
```
Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

---

**pandas.DataFrame.plot.hexbin**

`Dataframe.plot.hexbin(x, y, C=None, reduce_C_function=None, gridsize=None, **kwargs)`

Generate a hexagonal binning plot.

Generate a hexagonal binning plot of `x` versus `y`. If `C` is `None` (the default), this is a histogram of the number of occurrences of the observations at `(x[i], y[i])`.

If `C` is specified, specifies values at given coordinates `(x[i], y[i])`. These values are accumulated for each hexagonal bin and then reduced according to `reduce_C_function`, having as default the NumPy’s mean function (`numpy.mean()`). (If `C` is specified, it must also be a 1-D sequence of the same length as `x` and `y`, or a column label.)

**Parameters**

- `x` [int or str] The column label or position for `x` points.
- `y` [int or str] The column label or position for `y` points.
- `C` [int or str, optional] The column label or position for the value of `(x, y)` point.
- `reduce_C_function` [callable, default `np.mean`] Function of one argument that reduces all the values in a bin to a single number (e.g. `np.mean`, `np.max`, `np.sum`, `np.std`).
gridsize [int or tuple of (int, int), default 100] The number of hexagons in the x-direction.
The corresponding number of hexagons in the y-direction is chosen in a way that the
hexagons are approximately regular. Alternatively, gridsize can be a tuple with two
elements specifying the number of hexagons in the x-direction and the y-direction.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.AxesSubplot The matplotlib Axes on which the hexbin is plotted.

See also:

DataFrame.plot Make plots of a DataFrame.

matplotlib.pyplot.hexbin Hexagonal binning plot using matplotlib, the matplotlib function that is used
under the hood.

Examples

The following examples are generated with random data from a normal distribution.

```
>>> n = 10000
>>> df = pd.DataFrame({'x': np.random.randn(n),
...                      'y': np.random.randn(n)})
>>> ax = df.plot.hexbin(x='x', y='y', gridsize=20)
```
The next example uses `C` and `np.sum` as `reduce_C_function`. Note that `observations` values ranges from 1 to 5 but the result plot shows values up to more than 25. This is because of the `reduce_C_function`.

```python
>>> n = 500
>>> df = pd.DataFrame({
...     'coord_x': np.random.uniform(-3, 3, size=n),
...     'coord_y': np.random.uniform(30, 50, size=n),
...     'observations': np.random.randint(1,5, size=n)
...     })
>>> ax = df.plot.hexbin(x='coord_x',
...     y='coord_y',
...     C='observations',
...     reduce_C_function=np.sum,
...     gridsize=10,
...     cmap="viridis")
```
**pandas.DataFrame.plot.hist**

`DataFrame.plot.hist(by=None, bins=10, **kwargs)`

Draw one histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`. This is useful when the DataFrame's Series are in a similar scale.

**Parameters**

- `by` [str or sequence, optional] Column in the DataFrame to group by.
  
  Changed in version 1.4.0: Previously, `by` is silently ignore and makes no groupings

- `bins` [int, default 10] Number of histogram bins to be used.

- `**kwargs` Additional keyword arguments are documented in `DataFrame.plot()`.

**Returns**

- `class:matplotlib.AxesSubplot` Return a histogram plot.

**See also:**

- `DataFrame.hist` Draw histograms per DataFrame's Series.
- `Series.hist` Draw a histogram with Series' data.

**Examples**

When we roll a die 6000 times, we expect to get each value around 1000 times. But when we roll two dice and sum the result, the distribution is going to be quite different. A histogram illustrates those distributions.

```python
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns=['one'])

>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)

>>> ax = df.plot.hist(bins=12, alpha=0.5)
```

A grouped histogram can be generated by providing the parameter `by` (which can be a column name, or a list of column names):

```python
>>> age_list = [8, 10, 12, 14, 72, 74, 76, 78, 20, 25, 30, 35, 60, 85]

>>> df = pd.DataFrame({"gender": list("MMMMMMMMMMMMMM"), "age": age_list})

>>> ax = df.plot.hist(column="age", by="gender", figsize=(10, 8))
```

**pandas.DataFrame.plot.kde**

`DataFrame.plot.kde(bw_method=None, ind=None, **kwargs)`

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters**

- `bw_method` [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.
ind  [NumPy array or int, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If ind is a NumPy array, the KDE is evaluated at the points passed. If ind is an integer, ind number of equally spaced points are used.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or numpy.ndarray of them

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```python
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:
Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

For DataFrame, it works in the same way:

```python
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```python
>>> ax = df.plot.kde(bw_method=0.3)
>>> ax = df.plot.kde(bw_method=3)
```
2010 Chapter 3. API reference
3.4. DataFrame
Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```python
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

### pandas.DataFrame.plot.line

**DataFrame.plot.line**(x=None, y=None, **kwargs)

Plot Series or DataFrame as lines.

This function is useful to plot lines using DataFrame’s values as coordinates.

**Parameters**

- **x** [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.

- **y** [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

- **color** [str, array-like, or dict, optional] The color for each of the DataFrame’s columns. Possible values are:
  - A **single color string referred to by name**, RGB or RGBA code, for instance ‘red’ or ‘#a98d19’.
  - A **sequence of color strings referred to by name**, RGB or RGBA code, which will be used for each column recursively. For instance ['green','yellow'] each
column’s line will be filled in green or yellow, alternatively. If there is only a
single column to be plotted, then only the first color from the color list will be
used.

- A dict of the form {column name: color}, so that each column will be colored
  accordingly. For example, if your columns are called a and b, then passing
  {'a': 'green', 'b': 'red'} will color lines for column a in green and lines for
  column b in red.

New in version 1.1.0.

**kwargs Additional keyword arguments are documented in DataFrame.plot().

Returns

matplotlib.axes.Axes or np.ndarray of them An ndarray is returned with one
matplotlib.axes.Axes per column when subplots=True.

See also:

matplotlib.pyplot.plot Plot y versus x as lines and/or markers.

Examples

```python
>>> s = pd.Series([1, 3, 2])
>>> s.plot.line()
<AxesSubplot:ylabel='Density'>
```

The following example shows the populations for some animals over the years.

```python
>>> df = pd.DataFrame({'pig': [20, 18, 489, 675, 1776],
...                     'horse': [4, 25, 281, 600, 1900],
>>> lines = df.plot.line()
```

An example with subplots, so an array of axes is returned.

```python
>>> axes = df.plot.line(subplots=True)
>>> type(axes)
<class 'numpy.ndarray'>
```

Let’s repeat the same example, but specifying colors for each column (in this case, for each animal).

```python
>>> axes = df.plot.line(subplots=True, color={'pig': 'pink', 'horse': '#742802'})
```

The following example shows the relationship between both populations.

```python
>>> lines = df.plot.line(x='pig', y='horse')
```
3.4. DataFrame
**pandas.DataFrame.plot.pie**

DataFrame.plot.pie(**kwargs)

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps matplotlib.pyplot.pie() for the specified column. If no column reference is passed and subplots=True a pie plot is drawn for each numerical column independently.

**Parameters**

- y [int or label, optional] Label or position of the column to plot. If not provided, subplots=True argument must be passed.
- **kwargs Keyword arguments to pass on to DataFrame.plot().

**Returns**

- matplotlib.axes.Axes or np.ndarray of them A NumPy array is returned when subplots is True.

See also:

- Series.plot.pie Generate a pie plot for a Series.
- DataFrame.plot Make plots of a DataFrame.

**Examples**

In the example below we have a DataFrame with the information about planet’s mass and radius. We pass the 'mass' column to the pie function to get a pie plot.

```python
>>> df = pd.DataFrame({'mass': [0.330, 4.87 , 5.97],
...    'radius': [2439.7, 6051.8, 6378.1]},
...    index=['Mercury', 'Venus', 'Earth'])
>>> plot = df.plot.pie(y='mass', figsize=(5, 5))
```

```python
>>> plot = df.plot.pie(subplots=True, figsize=(11, 6))
```

**pandas.DataFrame.plot.scatter**

DataFrame.plot.scatter(x, y, s=None, c=None, **kwargs)

Create a scatter plot with varying marker point size and color.

The coordinates of each point are defined by two dataframe columns and filled circles are used to represent each point. This kind of plot is useful to see complex correlations between two variables. Points could be for instance natural 2D coordinates like longitude and latitude in a map or, in general, any pair of metrics that can be plotted against each other.

**Parameters**

- x [int or str] The column name or column position to be used as horizontal coordinates for each point.
- y [int or str] The column name or column position to be used as vertical coordinates for each point.
- s [str, scalar or array-like, optional] The size of each point. Possible values are:
  - A string with the name of the column to be used for marker’s size.
3.4. DataFrame
• A single scalar so all points have the same size.
• A sequence of scalars, which will be used for each point’s size recursively. For instance, when passing [2,14] all points size will be either 2 or 14, alternatively.

    Changed in version 1.1.0.

    c  [str, int or array-like, optional]  The color of each point. Possible values are:

• A single color string referred to by name, RGB or RGBA code, for instance ‘red’ or ‘#a98d19’.
• A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each point’s color recursively. For instance ['green','yellow'] all points will be filled in green or yellow, alternatively.
• A column name or position whose values will be used to color the marker points according to a colormap.

    **kwargs  Keyword arguments to pass on to DataFrame.plot().

Returns

    matplotlib.axes.Axes or numpy.ndarray of them

See also:

    matplotlib.pyplot.scatter  Scatter plot using multiple input data formats.

Examples

Let’s see how to draw a scatter plot using coordinates from the values in a DataFrame’s columns.

    >>> df = pd.DataFrame([[5.1, 3.5, 0], [4.9, 3.0, 0], [7.0, 3.2, 1],
    ... [6.4, 3.2, 1], [5.9, 3.0, 2]],
    ... columns=['length', 'width', 'species'])
    >>> ax1 = df.plot.scatter(x='length',
    ... y='width',
    ... c='DarkBlue')

    And now with the color determined by a column as well.

    >>> ax2 = df.plot.scatter(x='length',
    ... y='width',
    ... c='species',
    ... colormap='viridis')

    DataFrame.boxplot([column, by, ax, ...])  Make a box plot from DataFrame columns.
    DataFrame.hist([column, by, grid, ...])  Make a histogram of the DataFrame's columns.
3.4.16 Sparse accessor

Sparse-dtype specific methods and attributes are provided under the DataFrame.sparse accessor.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataFrame.sparse.density</strong></td>
<td>Ratio of non-sparse points to total (dense) data points.</td>
</tr>
<tr>
<td><strong>DataFrame.sparse.from_spmatrix</strong></td>
<td>Create a new DataFrame from a scipy sparse matrix.</td>
</tr>
<tr>
<td><strong>DataFrame.sparse.to_coo()</strong></td>
<td>Return the contents of the frame as a sparse SciPy COO matrix.</td>
</tr>
<tr>
<td><strong>DataFrame.sparse.to_dense()</strong></td>
<td>Convert a DataFrame with sparse values to dense.</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.sparse.density**

`DataFrame.sparse.density`

Ratio of non-sparse points to total (dense) data points.

**pandas.DataFrame.sparse.from_spmatrix**

classmethod `DataFrame.sparse.from_spmatrix(data, index=None, columns=None)`

Create a new DataFrame from a scipy sparse matrix.

- **Parameters**
  - `data` [scipy.sparse.spmatrix] Must be convertible to csc format.
  - `index, columns` [Index, optional] Row and column labels to use for the resulting DataFrame.
    - Defaults to a RangeIndex.

- **Returns**
  - `DataFrame` Each column of the DataFrame is stored as a arrays.SparseArray.

**Examples**

```python
>>> import scipy.sparse
>>> mat = scipy.sparse.eye(3)
>>> pd.DataFrame.sparse.from_spmatrix(mat)
   0  1  2
0  1.0 0.0 0.0
1  0.0 1.0 0.0
2  0.0 0.0 1.0
```
pandas.DataFrame.sparse.to_coo

DataFrame.sparse.to_coo()

Return the contents of the frame as a sparse SciPy COO matrix.

New in version 0.25.0.

Returns

coo_matrix  [scipy.sparse.spmatrix] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Notes

The dtype will be the lowest-common-denominator type (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. By numpy.find_common_type convention, mixing int64 and and uint64 will result in a float64 dtype.

pandas.DataFrame.sparse.to_dense

DataFrame.sparse.to_dense()

Convert a DataFrame with sparse values to dense.

New in version 0.25.0.

Returns

DataFrame  A DataFrame with the same values stored as dense arrays.

Examples

```python
>>> df = pd.DataFrame({'A': pd.arrays.SparseArray([0, 1, 0])})
>>> df.sparse.to_dense()
   A
0  0
1  1
2  0
```
3.17 Serialization / IO / conversion

- **DataFrame.from_dict**(data[, orient, dtype, ...]) Construct DataFrame from dict of array-like or dicts.
- **DataFrame.from_records**(data[, index, ...]) Convert structured or record ndarray to DataFrame.
- **DataFrame.to_parquet**(path[, engine, ...]) Write a DataFrame to the binary parquet format.
- **DataFrame.to_pickle**(path[, compression, ...]) Pickle (serialize) object to file.
- **DataFrame.to_csv**(path_or_buf, sep, na_rep, ...) Write object to a comma-separated values (csv) file.
- **DataFrame.to_hdf**(path_or_buf, key[, mode, ...]) Write the contained data to an HDF5 file using HDFS-tore.
- **DataFrame.to_sql**(name, con[, schema, ...]) Write records stored in a DataFrame to a SQL database.
- **DataFrame.to_excel**(excel_writer[, ...]) Write object to an Excel sheet.
- **DataFrame.to_json**(path_or_buf, orient, ...) Convert the object to a JSON string.
- **DataFrame.to_feather**(path, **kwargs) Write a DataFrame to the binary Feather format.
- **DataFrame.to_stata**(path[, convert_dates, ...]) Export DataFrame object to Stata dta format.
- **DataFrame.to_gbq**(destination_table[, ...]) Write a DataFrame to a Google BigQuery table.
- **DataFrame.to_records**(index, column_dtypes, ...]) Convert DataFrame to a NumPy record array.
- **DataFrame.to_string**(buf, columns, ...) Render a DataFrame to a console-friendly tabular output.
- **DataFrame.to_clipboard**(excel, sep) Copy object to the system clipboard.
- **DataFrame.to_markdown**(buf, mode, index, ...) Print DataFrame in Markdown-friendly format.
- **DataFrame.style** Returns a Styler object.

3.5 pandas arrays, scalars, and data types

For most data types, pandas uses NumPy arrays as the concrete objects contained with a Index, Series, or DataFrame.

For some data types, pandas extends NumPy's type system. String aliases for these types can be found at dtypes.

<table>
<thead>
<tr>
<th>Kind of Data</th>
<th>pandas Data Type</th>
<th>Scalar</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>TZ-aware datetime</td>
<td>DatetimeTZDtype</td>
<td>Timestamp</td>
<td>Datetime data</td>
</tr>
<tr>
<td>Timedeltas</td>
<td>(none)</td>
<td>Timedelta</td>
<td>Timedelta data</td>
</tr>
<tr>
<td>Period (time spans)</td>
<td>PeriodDtype</td>
<td>Period</td>
<td>Timespan data</td>
</tr>
<tr>
<td>Intervals</td>
<td>IntervalDtype</td>
<td>Interval</td>
<td>Interval data</td>
</tr>
<tr>
<td>Nullable Integer</td>
<td>Int64Dtype, ...</td>
<td>(none)</td>
<td>Nullable integer</td>
</tr>
<tr>
<td>Categorical</td>
<td>CategoricalDtype</td>
<td>(none)</td>
<td>Categorical data</td>
</tr>
<tr>
<td>Sparse</td>
<td>SparseDtype</td>
<td>(none)</td>
<td>Sparse data</td>
</tr>
<tr>
<td>Strings</td>
<td>StringDtype</td>
<td>str</td>
<td>Text data</td>
</tr>
<tr>
<td>Boolean (with NA)</td>
<td>BooleanDtype</td>
<td>bool</td>
<td>Boolean data with missing values</td>
</tr>
</tbody>
</table>

pandas and third-party libraries can extend NumPy’s type system (see Extension types). The top-level array() method can be used to create a new array, which may be stored in a Series, Index, or as a column in a DataFrame.

array(data[, dtype, copy]) Create an array.
3.5.1 pandas.array

The pandas.array function is used to create a new array.

**Parameters**

- `data`: A sequence of objects. These scalars inside `data` should be instances of the scalar type for `dtype`. It's expected that `data` represents a 1-dimensional array of data.

  When `data` is an Index or Series, the underlying array will be extracted from `data`.

- `dtype`: The dtype to use for the array. This may be a NumPy dtype or an extension type registered with pandas using `pandas.api.extensions.register_extension_dtype()`.

  If not specified, there are two possibilities:

  1. When `data` is a Series, Index, or ExtensionArray, the `dtype` will be taken from the data.
  2. Otherwise, pandas will attempt to infer the `dtype` from the data.

  Note that when `data` is a NumPy array, `data.dtype` is not used for inferring the array type. This is because NumPy cannot represent all the types of data that can be held in extension arrays.

  Currently, pandas will infer an extension dtype for sequences of strings.

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Array Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.Interval</td>
<td>pandas.arrays.IntervalArray</td>
</tr>
<tr>
<td>pandas.Period</td>
<td>pandas.arrays.PeriodArray</td>
</tr>
<tr>
<td>datetime.datetime</td>
<td>pandas.arrays.DatetimeArray</td>
</tr>
<tr>
<td>datetime.timedelta</td>
<td>pandas.arrays.TimedeltasArray</td>
</tr>
<tr>
<td>int</td>
<td>pandas.arrays.IntegerArray</td>
</tr>
<tr>
<td>float</td>
<td>pandas.arrays.FloatingArray</td>
</tr>
<tr>
<td>str</td>
<td>pandas.arrays.StringArray or pandas.arrays.ArrowStringArray</td>
</tr>
<tr>
<td>bool</td>
<td>pandas.arrays.BooleanArray</td>
</tr>
</tbody>
</table>

The ExtensionArray created when the scalar type is `str` is determined by `pd.options.mode.string_storage` if the dtype is not explicitly given.

For all other cases, NumPy’s usual inference rules will be used.

Changed in version 1.0.0: Pandas infers nullable-integer dtype for integer data, string dtype for string data, and nullable-boolean dtype for boolean data.

Changed in version 1.2.0: Pandas now also infers nullable-floating dtype for float-like input data.

- `copy`: A boolean, default True. Whether to copy the data, even if not necessary. Depending on the type of `data`, creating the new array may require copying data, even if `copy=False`.

**Returns**

- `ExtensionArray`: The newly created array.
Raises

**ValueError** When *data* is not 1-dimensional.

See also:

- `numpy.array` Construct a NumPy array.
- `Series` Construct a pandas Series.
- `Index` Construct a pandas Index.
- `arrays.PandasArray` ExtensionArray wrapping a NumPy array.
- `Series.array` Extract the array stored within a Series.

Notes

Omitting the `dtype` argument means pandas will attempt to infer the best array type from the values in the data. As new array types are added by pandas and 3rd party libraries, the “best” array type may change. We recommend specifying `dtype` to ensure that

1. the correct array type for the data is returned
2. the returned array type doesn’t change as new extension types are added by pandas and third-party libraries

Additionally, if the underlying memory representation of the returned array matters, we recommend specifying the `dtype` as a concrete object rather than a string alias or allowing it to be inferred. For example, a future version of pandas or a 3rd-party library may include a dedicated ExtensionArray for string data. In this event, the following would no longer return a `arrays.PandasArray` backed by a NumPy array.

```python
>>> pd.array(['a', 'b'], dtype=str)
<PandasArray>
['a', 'b']
Length: 2, dtype: str32
```

This would instead return the new ExtensionArray dedicated for string data. If you really need the new array to be backed by a NumPy array, specify that in the `dtype`.

```python
>>> pd.array(['a', 'b'], dtype=np.dtype("<U1"))
<PandasArray>
['a', 'b']
Length: 2, dtype: str32
```

Finally, Pandas has arrays that mostly overlap with NumPy

- `arrays.DatetimeArray`
- `arrays.TimedeltaArray`

When data with a `datetime64[ns]` or `timedelta64[ns]` dtype is passed, pandas will always return a `DatetimeArray` or `TimedeltaArray` rather than a `PandasArray`. This is for symmetry with the case of timezone-aware data, which NumPy does not natively support.

```python
>>> pd.array(['2015', '2016'], dtype='datetime64[ns]')
<DatetimeArray>
['2015-01-01 00:00:00', '2016-01-01 00:00:00']
Length: 2, dtype: datetime64[ns]
```

```python
>>> pd.array(['1H', '2H'], dtype='timedelta64[ns]')
<TimedeltaArray>
['0 days 01:00:00', '0 days 02:00:00']
Length: 2, dtype: timedelta64[ns]
```
Examples

If a dtype is not specified, pandas will infer the best dtype from the values. See the description of dtype for the types pandas infers for.

```python
>>> pd.array([1, 2])
<IntegerArray>
[1, 2]
Length: 2, dtype: Int64
```

```python
>>> pd.array([1, 2, np.nan])
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

```python
>>> pd.array([1.1, 2.2])
<FloatArray>
[1.1, 2.2]
Length: 2, dtype: Float64
```

```python
>>> pd.array(['a', None, 'c'])
<StringArray>
['a', <NA>, 'c']
Length: 3, dtype: string
```

```python
>>> with pd.option_context("string_storage", "pyarrow"):
...    arr = pd.array(['a', None, 'c'])
...    ...
```

```python
>>> arr
<ArrowStringArray>
['a', <NA>, 'c']
Length: 3, dtype: string
```

```python
<PeriodArray>
['2000-01-01', '2000-01-01']
Length: 2, dtype: period[D]
```

You can use the string alias for dtype

```python
>>> pd.array(['a', 'b', 'a'], dtype='category')
['a', 'b', 'a']
Categories (2, object): ['a', 'b']
```

Or specify the actual dtype

```python
>>> pd.array(['a', 'b', 'a'],
...           dtype=pd.CategoricalDtype(['a', 'b', 'c'], ordered=True))
['a', 'b', 'a']
Categories (3, object): ['a' < 'b' < 'c']
```

If pandas does not infer a dedicated extension type a `arrays.PandasArray` is returned.
>>> pd.array([1 + 1j, 3 + 2j])
<PandasArray>
[(1+1j), (3+2j)]
Length: 2, dtype: complex128

As mentioned in the “Notes” section, new extension types may be added in the future (by pandas or 3rd party libraries), causing the return value to no longer be a `arrays.PandasArray`. Specify the `dtype` as a NumPy `dtype` if you need to ensure there’s no future change in behavior.

```python
>>> pd.array([1, 2], dtype=np.dtype("int32"))
<PandasArray>
[1, 2]
Length: 2, dtype: int32
```

data must be 1-dimensional. A ValueError is raised when the input has the wrong dimensionality.

```python
>>> pd.array(1)
Traceback (most recent call last):
...
ValueError: Cannot pass scalar '1' to 'pandas.array'.
```

### 3.5.2 Datetime data

NumPy cannot natively represent timezone-aware datetimes. pandas supports this with the `arrays.DatetimeArray` extension array, which can hold timezone-naive or timezone-aware values.

`Timestamp`, a subclass of `datetime.datetime`, is pandas’ scalar type for timezone-naive or timezone-aware datetime data.

```python
Timestamp([ts_input, freq, tz, unit, year, ...]) Pandas replacement for python datetime.datetime object.
```

`pandas.Timestamp`

```python
class pandas.Timestamp(ts_input=<object object>, freq=None, tz=None, unit=None, year=None, month=None, day=None, hour=None, minute=None, second=None, microsecond=None, nanosecond=None, tzinfo=None, *, fold=None)
```

Pandas replacement for python datetime.datetime object.

Timestamp is the pandas equivalent of python’s Datetime and is interchangeable with it in most cases. It’s the type used for the entries that make up a DatetimeIndex, and other timeseries oriented data structures in pandas.

**Parameters**

- `ts_input` [datetime-like, str, int, float] Value to be converted to Timestamp.
- `freq` [str, DateOffset] Offset which Timestamp will have.
- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will have.
- `unit` [str] Unit used for conversion if `ts_input` is of type int or float. The valid values are ‘D’, ‘h’, ‘m’, ‘s’, ‘ms’, ‘us’, and ‘ns’. For example, ‘s’ means seconds and ‘ms’ means milliseconds.
year, month, day  [int]
hour, minute, second, microsecond  [int, optional, default 0]
nanosecond  [int, optional, default 0]
tzinfo  [datetime.tzinfo, optional, default None]
fold  [{0, 1}, default None, keyword-only] Due to daylight saving time, one wall clock time can occur twice when shifting from summer to winter time; fold describes whether the datetime-like corresponds to the first (0) or the second time (1) the wall clock hits the ambiguous time.
New in version 1.1.0.

Notes
There are essentially three calling conventions for the constructor. The primary form accepts four parameters. They can be passed by position or keyword.
The other two forms mimic the parameters from datetime.datetime. They can be passed by either position or keyword, but not both mixed together.

Examples
Using the primary calling convention:
This converts a datetime-like string

>>> pd.Timestamp('2017-01-01T12')
Timestamp('2017-01-01 12:00:00')

This converts a float representing a Unix epoch in units of seconds

>>> pd.Timestamp(1513393355.5, unit='s')
Timestamp('2017-12-16 03:02:35.500000')

This converts an int representing a Unix-epoch in units of seconds and for a particular timezone

>>> pd.Timestamp(1513393355, unit='s', tz='US/Pacific')
Timestamp('2017-12-15 19:02:35-0800', tz='US/Pacific')

Using the other two forms that mimic the API for datetime.datetime:

>>> pd.Timestamp(2017, 1, 1, 12)
Timestamp('2017-01-01 12:00:00')

>>> pd.Timestamp(year=2017, month=1, day=1, hour=12)
Timestamp('2017-01-01 12:00:00')
Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asm8</code></td>
<td>Return numpy datetime64 format in nanoseconds.</td>
</tr>
<tr>
<td><code>day_of_week</code></td>
<td>Return day of the week.</td>
</tr>
<tr>
<td><code>day_of_year</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>dayofweek</code></td>
<td>Return day of the week.</td>
</tr>
<tr>
<td><code>dayofyear</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>days_in_month</code></td>
<td>Return the number of days in the month.</td>
</tr>
<tr>
<td><code>daysinmonth</code></td>
<td>Return the number of days in the month.</td>
</tr>
<tr>
<td><code>freqstr</code></td>
<td>Return the total number of days in the month.</td>
</tr>
<tr>
<td><code>is_leap_year</code></td>
<td>Return True if year is a leap year.</td>
</tr>
<tr>
<td><code>is_month_end</code></td>
<td>Return True if date is last day of month.</td>
</tr>
<tr>
<td><code>is_month_start</code></td>
<td>Return True if date is first day of month.</td>
</tr>
<tr>
<td><code>is_quarter_end</code></td>
<td>Return True if date is last day of the quarter.</td>
</tr>
<tr>
<td><code>is_quarter_start</code></td>
<td>Return True if date is first day of the quarter.</td>
</tr>
<tr>
<td><code>is_year_end</code></td>
<td>Return True if date is last day of the year.</td>
</tr>
<tr>
<td><code>is_year_start</code></td>
<td>Return True if date is first day of the year.</td>
</tr>
<tr>
<td><code>quarter</code></td>
<td>Return the quarter of the year.</td>
</tr>
<tr>
<td><code>tz</code></td>
<td>Alias for tzinfo.</td>
</tr>
<tr>
<td><code>week</code></td>
<td>Return the week number of the year.</td>
</tr>
<tr>
<td><code>weekofyear</code></td>
<td>Return the week number of the year.</td>
</tr>
</tbody>
</table>

**pandas.Timestamp.asm8**

Timestamp.asm8

Return numpy datetime64 format in nanoseconds.

**Examples**

```python
>>> ts = pd.Timestamp(2020, 3, 14, 15)
>>> ts.asm8
numpy.datetime64('2020-03-14T15:00:00.000000000')
```

**pandas.Timestamp.day_of_week**

Timestamp.day_of_week

Return day of the week.

**Examples**

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.day_of_week
5
```
pandas.Timestamp.day_of_year

Timestamp.day_of_year
Return the day of the year.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.day_of_year
74
```

pandas.Timestamp.dayofweek

Timestamp.dayofweek
Return day of the week.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.day_of_week
5
```

pandas.Timestamp.dayofyear

Timestamp.dayofyear
Return the day of the year.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.day_of_year
74
```

pandas.Timestamp.days_in_month

Timestamp.days_in_month
Return the number of days in the month.
Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.days_in_month
31
```

**pandas.Timestamp.daysinmonth**

*Timestamp.daysinmonth*

Return the number of days in the month.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.days_in_month
31
```

**pandas.Timestamp.freqstr**

*property Timestamp.freqstr*

Return the total number of days in the month.

**pandas.Timestamp.is_leap_year**

*Timestamp.is_leap_year*

Return True if year is a leap year.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_leap_year
True
```

**pandas.Timestamp.is_month_end**

*Timestamp.is_month_end*

Return True if date is last day of month.
Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_month_end
False

>>> ts = pd.Timestamp(2020, 12, 31)
>>> ts.is_month_end
True
```

`pandas.Timestamp.is_month_end`

`Timestamp.is_month_end`

Return True if date is last day of month.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_month_end
False

>>> ts = pd.Timestamp(2020, 1, 1)
>>> ts.is_month_end
True
```

`pandas.Timestamp.is_quarter_end`

`Timestamp.is_quarter_end`

Return True if date is last day of the quarter.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_quarter_end
False

>>> ts = pd.Timestamp(2020, 3, 31)
>>> ts.is_quarter_end
True
```
pandas.Timestamp.is_quarter_start

Timestamp.is_quarter_start
Return True if date is first day of the quarter.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_quarter_start
False

>>> ts = pd.Timestamp(2020, 4, 1)
>>> ts.is_quarter_start
True
```

pandas.Timestamp.is_year_end

Timestamp.is_year_end
Return True if date is last day of the year.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_year_end
False

>>> ts = pd.Timestamp(2020, 12, 31)
>>> ts.is_year_end
True
```

pandas.Timestamp.is_year_start

Timestamp.is_year_start
Return True if date is first day of the year.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.is_year_start
False

>>> ts = pd.Timestamp(2020, 1, 1)
>>> ts.is_year_start
True
```
pandas.Timestamp.quarter

```
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.quarter
1
```

pandas.Timestamp.tz

```
>>> ts = pd.Timestamp(1584226800, unit='s', tz='Europe/Stockholm')
>>> ts.tz
<DstTzInfo 'Europe/Stockholm' CET+1:00:00 STD>
```

pandas.Timestamp.week

```
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.week
11
```

pandas.Timestamp.weekofyear

```
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.weekofyear
```

Return the week number of the year.
Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14)
>>> ts.week
11
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td></td>
</tr>
<tr>
<td>fold</td>
<td></td>
</tr>
<tr>
<td>freq</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td></td>
</tr>
<tr>
<td>microsecond</td>
<td></td>
</tr>
<tr>
<td>minute</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td></td>
</tr>
<tr>
<td>nanosecond</td>
<td></td>
</tr>
<tr>
<td>second</td>
<td></td>
</tr>
<tr>
<td>tzinfo</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td></td>
</tr>
</tbody>
</table>

Methods

- `astimezone(tz)`
  - Convert timezone-aware Timestamp to another timezone.
- `ceil([freq[, ambiguous, nonexistent]])`
  - Return a new Timestamp ceiled to this resolution.
- `combine(date, time)`
  - Combine date, time into datetime with same date and time fields.
- `ctime` (no `[, ]`)
  - Return ctime() style string.
- `date` (no `[, ]`)
  - Return date object with same year, month and day.
- `day_name`
  - Return the day name of the Timestamp with specified locale.
- `dst` (no `[, ]`)
  - Return self.tzinfo.dst(self).
- `floor([freq[, ambiguous, nonexistent]])`
  - Return a new Timestamp floored to this resolution.
- `fromisocalendar(int, int, int)`
  - Construct a date from the ISO year, week number and weekday.
- `fromisoformat(string)`
  - datetime from datetime.isoformat() output
- `fromordinal(ordinal[, freq, tz])`
  - Passed an ordinal, translate and convert to a ts.
- `fromtimestamp(ts)`
  - Transform timestamp[, tz] to tz's local time from POSIX timestamp.
- `isocalendar` (no `[, ]`)
  - Return a 3-tuple containing ISO year, week number, and weekday.
- `isoformat` (no `[, ]`)
  - Return the time formatted according to ISO 8610.
- `isoweekday()` (no `[, ]`)
  - Return the day of the week represented by the date.
- `month_name` (no `[, ]`)
  - Return the month name of the Timestamp with specified locale.
- `normalize` (no `[, ]`)
  - Normalize Timestamp to midnight, preserving tz information.
- `now([tz])` (no `[, ]`)
  - Return new Timestamp object representing current time local to tz.
- `replace([year, month, day, hour, minute, ...])`
  - Implements datetime.replace, handles nanoseconds.

continues on next page
Table 6 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round(freq[, ambiguous, nonexistent])</code></td>
<td>Round the Timestamp to the specified resolution.</td>
</tr>
<tr>
<td><code>strftime(format)</code></td>
<td>Return a string representing the given POSIX timestamp controlled by an explicit format string.</td>
</tr>
<tr>
<td><code>strptime(string, format)</code></td>
<td>Function is not implemented.</td>
</tr>
<tr>
<td><code>time(string, format)</code></td>
<td>Return time object with same time but with tz-info=None.</td>
</tr>
<tr>
<td><code>timestamp</code></td>
<td>Return POSIX timestamp as float.</td>
</tr>
<tr>
<td><code>timetuple</code></td>
<td>Return time tuple, compatible with time.localtime().</td>
</tr>
<tr>
<td><code>timetz</code></td>
<td>Return time object with same time and tzinfo.</td>
</tr>
<tr>
<td><code>to_datetime64</code></td>
<td>Return a numpy.datetime64 object with 'ns' precision.</td>
</tr>
<tr>
<td><code>to_julian_date()</code></td>
<td>Convert TimeStamp to a Julian Date.</td>
</tr>
<tr>
<td><code>to_numpy</code></td>
<td>Convert the Timestamp to a NumPy datetime64.</td>
</tr>
<tr>
<td><code>to_period</code></td>
<td>Return an period of which this timestamp is an observation.</td>
</tr>
<tr>
<td><code>to_pydatetime</code></td>
<td>Convert a Timestamp object to a native Python datetime object.</td>
</tr>
<tr>
<td><code>today(cls[, tz])</code></td>
<td>Return the current time in the local timezone.</td>
</tr>
<tr>
<td><code>toordinal</code></td>
<td>Return proleptic Gregorian ordinal.</td>
</tr>
<tr>
<td><code>tz_convert(tz)</code></td>
<td>Convert timezone-aware Timestamp to another time zone.</td>
</tr>
<tr>
<td><code>tz_localize(tz[, ambiguous, nonexistent])</code></td>
<td>Convert naive Timestamp to local time zone, or remove timezone from timezone-aware Timestamp.</td>
</tr>
<tr>
<td><code>tzname</code></td>
<td>Return self.tzinfo.tzname(self).</td>
</tr>
<tr>
<td><code>utcfromtimestamp(ts)</code></td>
<td>Construct a naive UTC datetime from a POSIX timestamp.</td>
</tr>
<tr>
<td><code>utcnow()</code></td>
<td>Return a new Timestamp representing UTC day and time.</td>
</tr>
<tr>
<td><code>utcoffset</code></td>
<td>Return self.tzinfo.utcoffset(self).</td>
</tr>
<tr>
<td><code>utctimetuple</code></td>
<td>Return UTC time tuple, compatible with time.localtime().</td>
</tr>
<tr>
<td><code>weekday()</code></td>
<td>Return the day of the week represented by the date.</td>
</tr>
</tbody>
</table>

**pandas.Timestamp.astimezone**

`Timestamp.astimezone(tz)`

Convert timezone-aware Timestamp to another time zone.

**Parameters**

- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

**Returns**

- `converted` [Timestamp]

**Raises**

- `TypeError` If Timestamp is tz-naive.
Examples

Create a timestamp object with UTC timezone:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651', tz='UTC')
>>> ts
Timestamp('2020-03-14 15:32:52.192548651+00:00', tz='UTC')
```

Change to Tokyo timezone:

```python
>>> ts.tz_convert(tz='Asia/Tokyo')
Timestamp('2020-03-15 00:32:52.192548651+09:00', tz='Asia/Tokyo')
```

Can also use `astimezone`:

```python
>>> ts.astimezone(tz='Asia/Tokyo')
Timestamp('2020-03-15 00:32:52.192548651+09:00', tz='Asia/Tokyo')
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.tz_convert(tz='Asia/Tokyo')
NaT
```

### pandas.Timestamp.ceil

`Timestamp.ceil(freq, ambiguous='raise', nonexistent='raise')`

Return a new Timestamp ceiled to this resolution.

**Parameters**

- `freq` [str] Frequency string indicating the ceiling resolution.
- `ambiguous` [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:
  - bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
  - ‘NaT’ will return NaT for an ambiguous time.
  - ‘raise’ will raise an AmbiguousTimeError for an ambiguous time.
- `nonexistent` [{‘raise’, ‘shift_forward’, ‘shift_backward’, ‘NaT’, timedelta}, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time.
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time.
  - ‘NaT’ will return NaT where there are nonexistent times.
  - timedelta objects will shift nonexistent times by the timedelta.
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Raises**

- `ValueError` if the freq cannot be converted.
Notes

If the Timestamp has a timezone, ceiling will take place relative to the local ("wall") time and re-localized to the same timezone. When ceiling near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

Examples

Create a timestamp object:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
```

A timestamp can be ceiled using multiple frequency units:

```python
>>> ts.ceil(freq='H')  # hour
Timestamp('2020-03-14 16:00:00')
```

```python
>>> ts.ceil(freq='T')  # minute
Timestamp('2020-03-14 15:33:00')
```

```python
>>> ts.ceil(freq='S')  # seconds
Timestamp('2020-03-14 15:32:53')
```

```python
>>> ts.ceil(freq='U')  # microseconds
Timestamp('2020-03-14 15:32:52.192549')
```

freq can also be a multiple of a single unit, like ‘5T’ (i.e. 5 minutes):

```python
>>> ts.ceil(freq='5T')
Timestamp('2020-03-14 15:35:00')
```

or a combination of multiple units, like ‘1H30T’ (i.e. 1 hour and 30 minutes):

```python
>>> ts.ceil(freq='1H30T')
Timestamp('2020-03-14 16:30:00')
```

Analogous for pd.NaT:

```python
>>> pd.NaT.ceil()
NaT
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> ts_tz = pd.Timestamp("2021-10-31 01:30:00").tz_localize("Europe/Amsterdam")
```

```python
>>> ts_tz.ceil("H", ambiguous=False)
Timestamp('2021-10-31 02:00:00+0100', tz='Europe/Amsterdam')
```

```python
>>> ts_tz.ceil("H", ambiguous=True)
Timestamp('2021-10-31 02:00:00+0200', tz='Europe/Amsterdam')
```
**pandas.Timestamp.combine**

**classmethod** `Timestamp.combine(date, time)`

Combine date, time into datetime with same date and time fields.

**Examples**

```python
>>> from datetime import date, time
>>> pd.Timestamp.combine(date(2020, 3, 14), time(15, 30, 15))
Timestamp('2020-03-14 15:30:15')
```

**pandas.Timestamp.ctime**

`Timestamp.ctime()`

Return ctime() style string.

**pandas.Timestamp.date**

`Timestamp.date()`

Return date object with same year, month and day.

**pandas.Timestamp.day_name**

`Timestamp.day_name()`

Return the day name of the Timestamp with specified locale.

**Parameters**

- `locale` [str, default None (English locale)] Locale determining the language in which to return the day name.

**Returns**

- `str`

**Examples**

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
>>> ts.day_name()
'Saturday'
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.day_name()
nan
```
pandas.Timestamp.dst

Timestamp.dst()
Return self.tzinfo.dst(self).

pandas.Timestamp.floor

Timestamp.floor(freq, ambiguous='raise', nonexistent='raise')
Return a new Timestamp floored to this resolution.

Parameters

freq [str] Frequency string indicating the flooring resolution.
ambiguous [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:
• bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
• ‘NaT’ will return NaT for an ambiguous time.
• ‘raise’ will raise an AmbiguousTimeError for an ambiguous time.
nonexistent [{‘raise’, ‘shift_forward’, ‘shift_backward’, ‘NaT’, timedelta}, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
• ‘shift_forward’ will shift the nonexistent time forward to the closest existing time.
• ‘shift_backward’ will shift the nonexistent time backward to the closest existing time.
• ‘NaT’ will return NaT where there are nonexistent times.
• timedelta objects will shift nonexistent times by the timedelta.
• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Raises

ValueError if the freq cannot be converted.

Notes

If the Timestamp has a timezone, flooring will take place relative to the local ("wall") time and re-localized to the same timezone. When flooring near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.
Examples

Create a timestamp object:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
```

A timestamp can be floored using multiple frequency units:

```python
>>> ts.floor(freq='H')  # hour
Timestamp('2020-03-14 15:00:00')
```

```python
>>> ts.floor(freq='T')  # minute
Timestamp('2020-03-14 15:32:00')
```

```python
>>> ts.floor(freq='S')  # seconds
Timestamp('2020-03-14 15:32:52')
```

```python
>>> ts.floor(freq='N')  # nanoseconds
Timestamp('2020-03-14 15:32:52.192548651')
```

`freq` can also be a multiple of a single unit, like ‘5T’ (i.e. 5 minutes):

```python
>>> ts.floor(freq='5T')
Timestamp('2020-03-14 15:30:00')
```

or a combination of multiple units, like ‘1H30T’ (i.e. 1 hour and 30 minutes):

```python
>>> ts.floor(freq='1H30T')
Timestamp('2020-03-14 15:00:00')
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.floor()
NaT
```

When rounding near a daylight savings time transition, use `ambiguous` or `nonexistent` to control how the timestamp should be re-localized.

```python
>>> ts_tz = pd.Timestamp("2021-10-31 03:30:00").tz_localize("Europe/Amsterdam")
```

```python
>>> ts_tz.floor("2H", ambiguous=False)
Timestamp('2021-10-31 02:00:00+0100', tz='Europe/Amsterdam')
```

```python
>>> ts_tz.floor("2H", ambiguous=True)
Timestamp('2021-10-31 02:00:00+0200', tz='Europe/Amsterdam')
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.Timestamp.fromisocalendar**

```python
timestamp = pandas.Timestamp.fromisocalendar(year, week, day)
```

**Parameters**
- `year` [int] ISO year
- `week` [int] Week number
- `day` [int] Weekday

Construct a date from the ISO year, week number and weekday.

This is the inverse of the date.isocalendar() function.

**pandas.Timestamp.fromisoformat**

```python
timestamp = pandas.Timestamp.fromisoformat(isoformat)
```

**Parameters**
- `isoformat` [str] Datetime formatted string

DateTime from datetime.isoformat() output.

**pandas.Timestamp.fromordinal**

```python
timestamp = pandas.Timestamp.fromordinal(ordinal, tz=None, freq=None)
```

**Parameters**
- `ordinal` [int] Date corresponding to a proleptic Gregorian ordinal.
- `freq` [str, DateOffset] Offset to apply to the Timestamp.
- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for the Timestamp.

Passed an ordinal, translate and convert to a ts. Note: by definition there cannot be any tz info on the ordinal itself.

**Examples**

```python
>>> pd.Timestamp.fromordinal(737425)
Timestamp('2020-01-01 00:00:00')
```

**pandas.Timestamp.fromtimestamp**

```python
timestamp = pandas.Timestamp.fromtimestamp(timestamp, tz=None)
```

**Parameters**
- `timestamp` [float] POSIX timestamp.
- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for the Timestamp.

Transform timestamp[, tz] to tz’s local time from POSIX timestamp.

**Examples**

```python
>>> pd.Timestamp.fromtimestamp(1584199972)
Timestamp('2020-03-14 15:32:52')
```

Note that the output may change depending on your local time.
pandas.Timestamp.isocalendar

**Timestamp.isocalendar()**
Return a 3-tuple containing ISO year, week number, and weekday.

pandas.Timestamp.isoformat

**Timestamp.isoformat()**
Return the time formatted according to ISO 8610.

The full format looks like ‘YYYY-MM-DD HH:MM:SS.mmmmmmmnn’. By default, the fractional part is omitted if self.microsecond == 0 and self.nanosecond == 0.

If self.tzinfo is not None, the UTC offset is also attached, giving a full format of ‘YYYY-MM-DD HH:MM:SS.mmmmmmmnn+HH:MM’.

**Parameters**
- `sep` [str, default ‘T’] String used as the separator between the date and time.
- `timespec` [str, default ‘auto’] Specifies the number of additional terms of the time to include. The valid values are ‘auto’, ‘hours’, ‘minutes’, ‘seconds’, ‘milliseconds’, ‘microseconds’, and ‘nanoseconds’.

**Returns**
- str

**Examples**

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
>>> ts.isoformat()
'2020-03-14T15:32:52.192548651'
>>> ts.isoformat(timespec='microseconds')
'2020-03-14T15:32:52.192548'
```

pandas.Timestamp.isoweekday

**Timestamp.isoweekday()**
Return the day of the week represented by the date. Monday == 1 ... Sunday == 7.

pandas.Timestamp.month_name

**Timestamp.month_name()**
Return the month name of the Timestamp with specified locale.

**Parameters**
- `locale` [str, default None (English locale)] Locale determining the language in which to return the month name.

**Returns**
- str
Examples

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
>>> ts.month_name()
'March'
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.month_name()
nan
```

**pandas.Timestamp.normalize**

`Timestamp.normalize()`

Normalize Timestamp to midnight, preserving tz information.

Examples

```python
>>> ts = pd.Timestamp(2020, 3, 14, 15, 30)
>>> ts.normalize()
Timestamp('2020-03-14 00:00:00')
```

**pandas.Timestamp.now**

`classmethod Timestamp.now(tz=None)`

Return new Timestamp object representing current time local to tz.

Parameters

- **tz** [str or timezone object, default None] Timezone to localize to.

Examples

```python
>>> pd.Timestamp.now()
Timestamp('2020-11-16 22:06:16.378782')
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.now()
NaT
```
pandas.Timestamp.replace

```
Timestamp.replace(year=None, month=None, day=None, hour=None, minute=None, second=None,
                   microsecond=None, nanosecond=None, tzinfo=<class 'object'>, fold=None)
```

Implements datetime.replace, handles nanoseconds.

**Parameters**

- `year` [int, optional]
- `month` [int, optional]
- `day` [int, optional]
- `hour` [int, optional]
- `minute` [int, optional]
- `second` [int, optional]
- `microsecond` [int, optional]
- `nanosecond` [int, optional]
- `tzinfo` [tz-convertible, optional]
- `fold` [int, optional]

**Returns**

Timestamp with fields replaced

**Examples**

Create a timestamp object:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651', tz='UTC')
>>> ts
Timestamp('2020-03-14 15:32:52.192548651+0000', tz='UTC')
```

Replace year and the hour:

```python
>>> ts.replace(year=1999, hour=10)
Timestamp('1999-03-14 10:32:52.192548651+0000', tz='UTC')
```

Replace timezone (not a conversion):

```python
>>> import pytz
>>> ts.replace(tzinfo=pytz.timezone('US/Pacific'))
Timestamp('2020-03-14 15:32:52.192548651-0700', tz='US/Pacific')
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.replace(tzinfo=pytz.timezone('US/Pacific'))
NaT
```
pandas.Timestamp.round

**pandas.Timestamp.round** *(freq, ambiguous='raise', nonexistent='raise')*

Round the Timestamp to the specified resolution.

**Parameters**

- **freq** [str] Frequency string indicating the rounding resolution.
- **ambiguous** [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:
  - bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
  - 'NaT' will return NaT for an ambiguous time.
  - 'raise' will raise an AmbiguousTimeError for an ambiguous time.
- **nonexistent** [{'raise', 'shift_forward', 'shift_backward', 'NaT', timedelta}, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - 'shift_forward' will shift the nonexistent time forward to the closest existing time.
  - 'shift_backward' will shift the nonexistent time backward to the closest existing time.
  - 'NaT' will return NaT where there are nonexistent times.
  - timedelta objects will shift nonexistent times by the timedelta.
  - 'raise' will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

A new Timestamp rounded to the given resolution of `freq`

**Raises**

ValueError if the freq cannot be converted

**Notes**

If the Timestamp has a timezone, rounding will take place relative to the local (“wall”) time and re-localized to the same timezone. When rounding near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

**Examples**

Create a timestamp object:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
```

A timestamp can be rounded using multiple frequency units:

```python
>>> ts.round(freq='H') # hour
Timestamp('2020-03-14 16:00:00')
```
```python
>>> ts.round(freq='T') # minute
Timestamp('2020-03-14 15:33:00')

>>> ts.round(freq='S') # seconds
Timestamp('2020-03-14 15:32:52')

>>> ts.round(freq='L') # milliseconds
Timestamp('2020-03-14 15:32:52.193000')

freq can also be a multiple of a single unit, like '5T' (i.e. 5 minutes):

```python
>>> ts.round(freq='5T')
Timestamp('2020-03-14 15:35:00')
```

or a combination of multiple units, like '1H30T' (i.e. 1 hour and 30 minutes):

```python
>>> ts.round(freq='1H30T')
Timestamp('2020-03-14 15:00:00')
```

Analogous for pd.NaT:

```python
>>> pd.NaT.round()
NaT
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> ts_tz = pd.Timestamp("2021-10-31 01:30:00").tz_localize("Europe/Amsterdam")

>>> ts_tz.round("H", ambiguous=False)
Timestamp('2021-10-31 02:00:00+0100', tz='Europe/Amsterdam')

>>> ts_tz.round("H", ambiguous=True)
Timestamp('2021-10-31 02:00:00+0200', tz='Europe/Amsterdam')
```

**pandas.Timestamp.strftime**

`Timestamp.strftime(format)`

Return a string representing the given POSIX timestamp controlled by an explicit format string.

**Parameters**

- **format** [str] Format string to convert Timestamp to string. See strftime documentation for more information on the format string: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.
Examples

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
>>> ts.strftime('%Y-%m-%d %X')
'2020-03-14 15:32:52'
```

**pandas.Timestamp.strptime**

*classmethod* Timestamp.strptime(string, format)  
Function is not implemented. Use pd.to_datetime().

**pandas.Timestamp.time**

Timestamp.time()  
Return time object with same time but with tzinfo=None.

**pandas.Timestamp.timestamp**

Timestamp.timestamp()  
Return POSIX timestamp as float.

Examples

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548')
>>> ts.timestamp()
1584199972.192548
```

**pandas.Timestamp.timetuple**

Timestamp.timetuple()  
Return time tuple, compatible with time.localtime().

**pandas.Timestamp.timetz**

Timestamp.timetz()  
Return time object with same time and tzinfo.
**pandas.Timestamp.to_datetime64**

`Timestamp.to_datetime64()`

Return a numpy.datetime64 object with ‘ns’ precision.

**pandas.Timestamp.to_julian_date**

`Timestamp.to_julian_date()`

Convert `Time`Stamp to a Julian Date. 0 Julian date is noon January 1, 4713 BC.

**Examples**

```python
gt = pd.Timestamp('2020-03-14 15:32:52')
gt.to_julian_date()  # 2458923.147824074
```

**pandas.Timestamp.to_numpy**

`Timestamp.to_numpy()`

Convert the `Time`stamp to a NumPy datetime64.

New in version 0.25.0.

This is an alias method for `Timestamp.to_datetime64()`. The dtype and copy parameters are available here only for compatibility. Their values will not affect the return value.

**Returns**

`numpy.datetime64`

**See also:**

`DatetimeIndex.to_numpy` Similar method for DatetimeIndex.

**Examples**

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548651')
gt.to_numpy()  # numpy.datetime64('2020-03-14T15:32:52.192548651')

Analogous for `pd.NaT`:

```python
gt = pd.NaT.to_numpy()  # numpy.datetime64('NaT')
```
**pandas.Timestamp.to_period**

**Timestamp.to_period()**

Return an period of which this timestamp is an observation.

**Examples**

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548651')
print(gt.to_period(freq='Y'))
Period('2020', 'A-DEC')
```

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548651')
print(gt.to_period(freq='M'))
Period('2020-03', 'M')
```

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548651')
print(gt.to_period(freq='W'))
Period('2020-03-09/2020-03-15', 'W-SUN')
```

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548651')
print(gt.to_period(freq='Q'))
Period('2020Q1', 'Q-DEC')
```

**pandas.Timestamp.to_pydatetime**

**Timestamp.to_pydatetime()**

Convert a Timestamp object to a native Python datetime object.

If warn=True, issue a warning if nanoseconds is nonzero.

**Examples**

```python
gt = pd.Timestamp('2020-03-14T15:32:52.192548')
gt.to_pydatetime()
datetime.datetime(2020, 3, 14, 15, 32, 52, 192548)
```

Analogous for pd.NaT:

```python
print(pd.NaT.to_pydatetime())
NaT
```
pandas.Timestamp.today

classmethod Timestamp.today(cls, tz=None)

Return the current time in the local timezone. This differs from datetime.today() in that it can be localized to a passed timezone.

Parameters

tz [str or timezone object, default None] Timezone to localize to.

Examples

```python
>>> pd.Timestamp.today()
Timestamp('2020-11-16 22:37:39.969883')
```

Analogous for pd.NaT:

```python
>>> pd.NaT.today()
NaT
```

pandas.Timestamp.toordinal

Timestamp.toordinal()

Return proleptic Gregorian ordinal. January 1 of year 1 is day 1.

pandas.Timestamp.tz_convert

Timestamp.tz_convert(tz)

Convert timezone-aware Timestamp to another time zone.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

Returns

class [Timestamp]

Raises

TypeError If Timestamp is tz-naive.

Examples

Create a timestamp object with UTC timezone:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651', tz='UTC')
>>> ts
Timestamp('2020-03-14 15:32:52.192548651+0000', tz='UTC')
```

Change to Tokyo timezone:
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> ts.tz_convert(tz='Asia/Tokyo')
Timestamp('2020-03-15 00:32:52.192548651+0900', tz='Asia/Tokyo')
```

Can also use `astimezone`

```python
>>> ts.astimezone(tz='Asia/Tokyo')
Timestamp('2020-03-15 00:32:52.192548651+0900', tz='Asia/Tokyo')
```

Analogous for `pd.NaT`

```python
>>> pd.NaT.tz_convert(tz='Asia/Tokyo')
NaT
```

**pandas.Timestamp.tz_localize**

`Timestamp.tz_localize(tz, ambiguous='raise', nonexistent='raise')`

Convert naive Timestamp to local time zone, or remove timezone from timezone-aware Timestamp.

**Parameters**

- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding local time.

- `ambiguous` [bool, ‘NaT’, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the `ambiguous` parameter dictates how ambiguous times should be handled.

  - The behavior is as follows:
    - `bool` contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
    - ‘NaT’ will return NaT for an ambiguous time.
    - ‘raise’ will raise an `AmbiguousTimeError` for an ambiguous time.

- `nonexistent` ['shift_forward', 'shift_backward', ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

  - The behavior is as follows:
    - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time.
    - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time.
    - ‘NaT’ will return NaT where there are nonexistent times.
    - `timedelta` objects will shift nonexistent times by the `timedelta`.
    - ‘raise’ will raise an `NonExistentTimeError` if there are nonexistent times.

**Returns**

- `localized` [Timestamp]

**Raises**
**TypeError** If the Timestamp is tz-aware and tz is not None.

### Examples

Create a naive timestamp object:

```python
>>> ts = pd.Timestamp('2020-03-14T15:32:52.192548651')
>>> ts
Timestamp('2020-03-14 15:32:52.192548651')
```

Add ‘Europe/Stockholm’ as timezone:

```python
>>> ts.tz_localize(tz='Europe/Stockholm')
Timestamp('2020-03-14 15:32:52.192548651+0100', tz='Europe/Stockholm')
```

Analogous for `pd.NaT`:

```python
>>> pd.NaT.tz_localize()
NaT
```

**pandas.Timestamp.tzname**

Timestamp.tzname()  
Return self.tzinfo.tzname(self).

**pandas.Timestamp.utcfromtimestamp**

`classmethod` Timestamp.utcfromtimestamp(ts)  
Construct a naive UTC datetime from a POSIX timestamp.

### Examples

```python
>>> pd.Timestamp.utcfromtimestamp(1584199972)
Timestamp('2020-03-14 15:32:52')
```

**pandas.Timestamp.utcnow**

`classmethod` Timestamp.utcnow()  
Return a new Timestamp representing UTC day and time.
Examples

```python
>>> pd.Timestamp.utcnow()
Timestamp('2020-11-16 22:50:18.092888+0000', tz='UTC')
```

**pandas.Timestamp.utcoffset**

`Timestamp.utcoffset()`  
Return `self.tzinfo.utcoffset(self)`.

**pandas.Timestamp.utctimetuple**

`Timestamp.utctimetuple()`  
Return UTC time tuple, compatible with `time.localtime()`.

**pandas.Timestamp.weekday**

`Timestamp.weekday()`  
Return the day of the week represented by the date. Monday == 0 ... Sunday == 6.

**Properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Timestamp.asn8</code></td>
<td>Return numpy datetime64 format in nanoseconds.</td>
</tr>
<tr>
<td><code>Timestamp.day</code></td>
<td></td>
</tr>
<tr>
<td><code>Timestamp.dayofweek</code></td>
<td>Return day of the week.</td>
</tr>
<tr>
<td><code>Timestamp.day_of_week</code></td>
<td>Return day of the week.</td>
</tr>
<tr>
<td><code>Timestamp.dayofyear</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>Timestamp.day_of_year</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>Timestamp.days_in_month</code></td>
<td>Return the number of days in the month.</td>
</tr>
<tr>
<td><code>Timestamp.daysinmonth</code></td>
<td>Return the number of days in the month.</td>
</tr>
<tr>
<td><code>Timestamp.fold</code></td>
<td></td>
</tr>
<tr>
<td><code>Timestamp.hour</code></td>
<td>Return True if year is a leap year.</td>
</tr>
<tr>
<td><code>Timestamp.is_leap_year</code></td>
<td>Return True if year is a leap year.</td>
</tr>
<tr>
<td><code>Timestamp.is_month_end</code></td>
<td>Return True if date is last day of month.</td>
</tr>
<tr>
<td><code>Timestamp.is_month_start</code></td>
<td>Return True if date is first day of month.</td>
</tr>
<tr>
<td><code>Timestamp.is_quarter_end</code></td>
<td>Return True if date is last day of the quarter.</td>
</tr>
<tr>
<td><code>Timestamp.is_quarter_start</code></td>
<td>Return True if date is first day of the quarter.</td>
</tr>
<tr>
<td><code>Timestamp.is_year_end</code></td>
<td>Return True if date is last day of the year.</td>
</tr>
<tr>
<td><code>Timestamp.is_year_start</code></td>
<td>Return True if date is first day of the year.</td>
</tr>
<tr>
<td><code>Timestamp.max</code></td>
<td></td>
</tr>
<tr>
<td><code>Timestamp.microsecond</code></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp.min</td>
<td></td>
</tr>
<tr>
<td>Timestamp.minute</td>
<td></td>
</tr>
<tr>
<td>Timestamp.month</td>
<td></td>
</tr>
<tr>
<td>Timestamp.nanosecond</td>
<td></td>
</tr>
<tr>
<td>Timestamp.quarter</td>
<td>Return the quarter of the year.</td>
</tr>
<tr>
<td>Timestamp.resolution</td>
<td></td>
</tr>
<tr>
<td>Timestamp.second</td>
<td></td>
</tr>
<tr>
<td>Timestamp.tz</td>
<td>Alias for tzinfo.</td>
</tr>
<tr>
<td>Timestamp.tzinfo</td>
<td></td>
</tr>
<tr>
<td>Timestamp.value</td>
<td></td>
</tr>
<tr>
<td>Timestamp.week</td>
<td>Return the week number of the year.</td>
</tr>
<tr>
<td>Timestamp.weekofyear</td>
<td>Return the week number of the year.</td>
</tr>
<tr>
<td>Timestamp.year</td>
<td></td>
</tr>
</tbody>
</table>

pandas.Timestamp.day

Timestamp.day

pandas.Timestamp.fold

Timestamp.fold

pandas.Timestamp.hour

Timestamp.hour

pandas.Timestamp.max

Timestamp.max = Timestamp('2262-04-11 23:47:16.854775807')
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.Timestamp.microsecond

Timestamp.microsecond

pandas.Timestamp.min

Timestamp.min = Timestamp('1677-09-21 00:12:43.145224193')

pandas.Timestamp.minute

Timestamp.minute

pandas.Timestamp.month

Timestamp.month

pandas.Timestamp.nanosecond

Timestamp.nanosecond

pandas.Timestamp.resolution

Timestamp.resolution = Timedelta('0 days 00:00:00.000000001')

pandas.Timestamp.second

Timestamp.second

pandas.Timestamp.tzinfo

Timestamp.tzinfo

pandas.Timestamp.value

Timestamp.value
pandas.Timestamp.year

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp.astimezone(tz)</td>
<td>Convert timezone-aware Timestamp to another timezone.</td>
</tr>
<tr>
<td>Timestamp.ceil(freq[, ambiguous, nonexistent])</td>
<td>Return a new Timestamp ceiled to this resolution.</td>
</tr>
<tr>
<td>Timestamp.combine(date, time)</td>
<td>Combine date, time into datetime with same date and time fields.</td>
</tr>
<tr>
<td>Timestampctime</td>
<td>Return ctime() style string.</td>
</tr>
<tr>
<td>Timestamp.date</td>
<td>Return date object with same year, month and day.</td>
</tr>
<tr>
<td>Timestamp.day_name</td>
<td>Return the day name of the Timestamp with specified locale.</td>
</tr>
<tr>
<td>Timestamp.dst</td>
<td>Return self.tzinfo.dst(self).</td>
</tr>
<tr>
<td>Timestamp.floor(freq[, ambiguous, nonexistent])</td>
<td>Return a new Timestamp floored to this resolution.</td>
</tr>
<tr>
<td>Timestamp.freq</td>
<td></td>
</tr>
<tr>
<td>Timestamp.freqstr</td>
<td>Return the total number of days in the month.</td>
</tr>
<tr>
<td>Timestamp.fromordinal(ordinal[, freq, tz])</td>
<td>Passed an ordinal, translate and convert to a ts.</td>
</tr>
<tr>
<td>Timestamp.fromtimestamp(ts)</td>
<td>Transform timestamp[, tz] to tz's local time from POSIX timestamp.</td>
</tr>
<tr>
<td>Timestamp.isocalendar</td>
<td>Return a 3-tuple containing ISO year, week number, and weekday.</td>
</tr>
<tr>
<td>Timestamp.isoformat</td>
<td>Return the time formatted according to ISO 8610.</td>
</tr>
<tr>
<td>Timestamp.isoweekday</td>
<td>Return the day of the week represented by the date.</td>
</tr>
<tr>
<td>Timestamp.month_name</td>
<td>Return the month name of the Timestamp with specified locale.</td>
</tr>
<tr>
<td>Timestamp.normalize</td>
<td>Normalize Timestamp to midnight, preserving tz information.</td>
</tr>
<tr>
<td>Timestamp.now([tz])</td>
<td>Return new Timestamp object representing current time local to tz.</td>
</tr>
<tr>
<td>Timestamp.replace([year, month, day, hour, ...])</td>
<td>Implements datetime.replace, handles nanoseconds.</td>
</tr>
<tr>
<td>Timestamp.round(freq[, ambiguous, nonexistent])</td>
<td>Round the Timestamp to the specified resolution.</td>
</tr>
<tr>
<td>Timestamp.strftime(format)</td>
<td>Return a string representing the given POSIX timestamp controlled by an explicit format string.</td>
</tr>
<tr>
<td>Timestamp.strptime(string, format)</td>
<td>Function is not implemented.</td>
</tr>
<tr>
<td>Timestamp.time</td>
<td>Return time object with same time but with tzinfo=None.</td>
</tr>
<tr>
<td>Timestamp.timestamp</td>
<td>Return POSIX timestamp as float.</td>
</tr>
<tr>
<td>Timestamp.timetuple</td>
<td>Return time tuple, compatible with time.localtime().</td>
</tr>
<tr>
<td>Timestamp.timetz</td>
<td>Return time object with same time and tzinfo.</td>
</tr>
<tr>
<td>Timestamp.to_datetime64</td>
<td>Return a numpy.datetime64 object with 'ns' precision.</td>
</tr>
<tr>
<td>Timestamp.to_period</td>
<td>Return a period of which this timestamp is an observation.</td>
</tr>
<tr>
<td>Timestamp.to_pydatetime</td>
<td>Convert a Timestamp object to a native Python datetime object.</td>
</tr>
<tr>
<td>Timestamp.today(cls[, tz])</td>
<td>Return the current time in the local timezone.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Timestamp.toordinal</code></td>
<td>Return proleptic Gregorian ordinal.</td>
</tr>
<tr>
<td><code>Timestamp.tz_convert(tz)</code></td>
<td>Convert timezone-aware Timestamp to another timezone.</td>
</tr>
<tr>
<td><code>Timestamp.tz_localize(tz[, ambiguous, ...])</code></td>
<td>Convert naive Timestamp to local time zone, or remove timezone from timezone-aware Timestamp.</td>
</tr>
<tr>
<td><code>Timestamp.tzname</code></td>
<td>Return self.tzinfo.tzname(self).</td>
</tr>
<tr>
<td><code>Timestamp.utcfromtimestamp(ts)</code></td>
<td>Construct a naive UTC datetime from a POSIX timestamp.</td>
</tr>
<tr>
<td><code>Timestamp.utcnow()</code></td>
<td>Return a new Timestamp representing UTC day and time.</td>
</tr>
<tr>
<td><code>Timestamp.utcoffset</code></td>
<td>Return self.tzinfo.utcoffset(self).</td>
</tr>
<tr>
<td><code>Timestamp.utctimetuple</code></td>
<td>Return UTC time tuple, compatible with <code>time.localtime()</code>.</td>
</tr>
<tr>
<td><code>Timestamp.weekday()</code></td>
<td>Return the day of the week represented by the date.</td>
</tr>
</tbody>
</table>

### pandas.Timestamp.freq

A collection of timestamps may be stored in a `arrays.DatetimeArray`. For timezone-aware data, the `.dtype` of a `arrays.DatetimeArray` is a `DatetimeTZDtype`. For timezone-naive data, `np.dtype("datetime64[ns]")` is used.

If the data are timezone-aware, then every value in the array must have the same timezone.

```python
arrays.DatetimeArray(values[, dtype, freq, copy])
```
Pandas ExtensionArray for tz-naive or tz-aware datetime data.

### pandas.arrays.DatetimeArray

**class** `pandas.arrays.DatetimeArray(values[, dtype, freq, copy])`

Pandas ExtensionArray for tz-naive or tz-aware datetime data.

**Warning:** DatetimeArray is currently experimental, and its API may change without warning. In particular, `DatetimeArray.dtype` is expected to change to always be an instance of an `ExtensionDtype` subclass.

**Parameters**

- `values` [Series, Index, DatetimeArray, ndarray] The datetime data.
  
  For DatetimeArray `values` (or a Series or Index boxing one), `dtype` and `freq` will be extracted from `values`.

- `dtype` [numpy.dtype or DatetimeTZDtype] Note that the only NumPy dtype allowed is ‘datetime64[ns]’.

- `freq` [str or Offset, optional] The frequency.

- `copy` [bool, default False] Whether to copy the underlying array of values.
Attributes

None

Methods

None

DatetimeTZDtype([unit, tz])

An ExtensionDtype for timezone-aware datetime data.

pandas.DatetimeTZDtype

class pandas.DatetimeTZDtype(unit='ns', tz=None)

An ExtensionDtype for timezone-aware datetime data.

This is not an actual numpy dtype, but a duck type.

Parameters

- **unit** [str, default "ns"] The precision of the datetime data. Currently limited to "ns".
- **tz** [str, int, or datetime.tzinfo] The timezone.

Raises

- `pytz.UnknownTimeZoneError` When the requested timezone cannot be found.

Examples

```python
>>> pd.DatetimeTZDtype(tz='UTC')
datetime64[ns, UTC]
```

```python
>>> pd.DatetimeTZDtype(tz='dateutil/US/Central')
datetime64[ns, tzfile('/usr/share/zoneinfo/US/Central')]
```

Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unit</strong></td>
<td>The precision of the datetime data.</td>
</tr>
<tr>
<td><strong>tz</strong></td>
<td>The timezone.</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DatetimeTZDtype.unit**

*property* `DatetimeTZDtype.unit`

The precision of the datetime data.

**pandas.DatetimeTZDtype.tz**

*property* `DatetimeTZDtype.tz`

The timezone.

**Methods**

```
None
```

### 3.5.3 Timedelta data

NumPy can natively represent timedeltas. pandas provides `Timedelta` for symmetry with `Timestamp`.

```
Timedelta([value, unit])
```

Represents a duration, the difference between two dates or times.

**pandas.Timedelta**

```
class pandas.Timedelta(value=<object object>, unit=None, **kwargs)
```

Represents a duration, the difference between two dates or times.

Timedelta is the pandas equivalent of python’s `datetime.timedelta` and is interchangeable with it in most cases.

**Parameters**

- **value** [Timedelta, timedelta, np.timedelta64, str, or int]
- **unit** [str, default ‘ns’] Denote the unit of the input, if input is an integer.

Possible values:

- ‘days’ or ‘day’
- ‘hours’, ‘hour’, ‘hr’, or ‘h’
- ‘minutes’, ‘minute’, ‘min’, or ‘m’
- ‘seconds’, ‘second’, or ‘sec’
- ‘milliseconds’, ‘millisecond’, ‘millis’, or ‘milli’
- ‘microseconds’, ‘microsecond’, ‘micros’, or ‘micro’

**kwargs** Available kwargs: {days, seconds, microseconds, milliseconds, minutes, hours, weeks}. Values for construction in compat with datetime.timedelta. Numpy ints and floats will be coerced to python ints and floats.
Notes

The constructor may take in either both values of value and unit or kwargs as above. Either one of them must be used during initialization.

The `.value` attribute is always in ns.

If the precision is higher than nanoseconds, the precision of the duration is truncated to nanoseconds.

Examples

Here we initialize Timedelta object with both value and unit

```python
>>> td = pd.Timedelta(1, "d")
>>> td
Timedelta('1 days 00:00:00')
```

Here we initialize the Timedelta object with kwargs

```python
>>> td2 = pd.Timedelta(days=1)
>>> td2
Timedelta('1 days 00:00:00')
```

We see that either way we get the same result

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asm8</code></td>
<td>Return a numpy timedelta64 array scalar view.</td>
</tr>
<tr>
<td><code>components</code></td>
<td>Return a components namedtuple-like.</td>
</tr>
<tr>
<td><code>days</code></td>
<td>Number of days.</td>
</tr>
<tr>
<td><code>delta</code></td>
<td>Return the timedelta in nanoseconds (ns), for internal compatibility.</td>
</tr>
<tr>
<td><code>microseconds</code></td>
<td>Number of microseconds (&gt;= 0 and less than 1 second).</td>
</tr>
<tr>
<td><code>nanoseconds</code></td>
<td>Return the number of nanoseconds (n), where 0 &lt;= n &lt; 1 microsecond.</td>
</tr>
<tr>
<td><code>resolution_string</code></td>
<td>Return a string representing the lowest timedelta resolution.</td>
</tr>
<tr>
<td><code>seconds</code></td>
<td>Number of seconds (&gt;= 0 and less than 1 day).</td>
</tr>
</tbody>
</table>

`pandas.Timedelta.asm8`

Timedelta.asm8

Return a numpy timedelta64 array scalar view.

Provides access to the array scalar view (i.e. a combination of the value and the units) associated with the numpy.timedelta64().view(), including a 64-bit integer representation of the timedelta in nanoseconds (Python int compatible).

Returns

- **numpy timedelta64 array scalar view** Array scalar view of the timedelta in nanoseconds.
### Examples

```python
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
```
```
>>> td.asm8
numpy.timedelta64(8652000003042, 'ns')
```

```python
>>> td = pd.Timedelta('2 min 3 s')
```
```
>>> td.asm8
numpy.timedelta64(123000000000, 'ns')
```

```python
>>> td = pd.Timedelta('3 ms 5 us')
```
```
>>> td.asm8
numpy.timedelta64(3005000, 'ns')
```

```python
>>> td = pd.Timedelta(42, unit='ns')
```
```
>>> td.asm8
numpy.timedelta64(42, 'ns')
```

### pandas.Timedelta.components

**Timedelta.components**

Return a components namedtuple-like.

### pandas.Timedelta.days

**Timedelta.days**

Number of days.

### pandas.Timedelta.delta

**Timedelta.delta**

Return the timedelta in nanoseconds (ns), for internal compatibility.

**Returns**

| int | Timedelta in nanoseconds. |

### Examples

```python
>>> td = pd.Timedelta('1 days 42 ns')
```
```
>>> td.delta
86400000000042
```

```python
>>> td = pd.Timedelta('3 s')
```
```
>>> td.delta
3000000000
```
```python
>>> td = pd.Timedelta('3 ms 5 us')
>>> td.delta
3005000

>>> td = pd.Timedelta(42, unit='ns')
>>> td.delta
42
```

### pandas.Timedelta.microseconds

**Timedelta.microseconds**

Number of microseconds (>= 0 and less than 1 second).

### pandas.Timedelta.nanoseconds

**Timedelta.nanoseconds**

Return the number of nanoseconds (n), where 0 <= n < 1 microsecond.

**Returns**

- `int` Number of nanoseconds.

**See also:**

- `Timedelta.components` Return all attributes with assigned values (i.e. days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds).

### Examples

**Using string input**

```python
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
```

```python
>>> td.nanoseconds
42
```

**Using integer input**

```python
>>> td = pd.Timedelta(42, unit='ns')
```

```python
>>> td.nanoseconds
42
```
pandas.Timedelta.resolution_string

Timedelta.resolution_string

Return a string representing the lowest timedelta resolution.

Each timedelta has a defined resolution that represents the lowest OR most granular level of precision. Each level of resolution is represented by a short string as defined below:

Resolution: Return value

- Days: ‘D’
- Hours: ‘H’
- Minutes: ‘T’
- Seconds: ‘S’
- Milliseconds: ‘L’
- Microseconds: ‘U’
- Nanoseconds: ‘N’

Returns

str  Timedelta resolution.

Examples

```python
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
>>> td.resolution_string
'N'

>>> td = pd.Timedelta('1 days 2 min 3 us')
>>> td.resolution_string
'U'

>>> td = pd.Timedelta('2 min 3 s')
>>> td.resolution_string
'S'

>>> td = pd.Timedelta(36, unit='us')
>>> td.resolution_string
'U'
```
**pandas.Timedelta.seconds**

**Timedelta.seconds**

Number of seconds (>= 0 and less than 1 day).

<table>
<thead>
<tr>
<th>freq</th>
<th>is_populated</th>
<th>value</th>
</tr>
</thead>
</table>

**Methods**

- **ceil(freq)**
  Return a new Timedelta ceiled to this resolution.
- **floor(freq)**
  Return a new Timedelta floored to this resolution.
- **isoformat**
- **round(freq)**
  Round the Timedelta to the specified resolution.
- **to_numpy**
  Convert the Timedelta to a NumPy timedelta64.
- **to_pytimedelta**
  Convert a pandas Timedelta object into a python datetime.timedelta object.
- **to_timedelta64**
  Return a numpy.timedelta64 object with 'ns' precision.
- **total_seconds**
  Total seconds in the duration.
- **view**
  Array view compatibility.

**pandas.Timedelta.ceil**

**Timedelta.ceil(freq)**

Return a new Timedelta ceiled to this resolution.

**Parameters**

- **freq** [str] Frequency string indicating the ceiling resolution.

**pandas.Timedelta.floor**

**Timedelta.floor(freq)**

Return a new Timedelta floored to this resolution.

**Parameters**

- **freq** [str] Frequency string indicating the flooring resolution.
### pandas.Timedelta.isoformat

**Timedelta.isoformat()**


**Returns**

`str`

**See also:**

**Timestamp.isoformat** Function is used to convert the given Timestamp object into the ISO format.

**Notes**

The longest component is days, whose value may be larger than 365. Every component is always included, even if its value is 0. Pandas uses nanosecond precision, so up to 9 decimal places may be included in the seconds component. Trailing 0's are removed from the seconds component after the decimal. We do not 0 pad components, so it's ...`TSH`... not ...`T0SH`...

**Examples**

```python
>>> td = pd.Timedelta(days=6, minutes=50, seconds=3,
...                   milliseconds=10, microseconds=10, nanoseconds=12)
>>> td.isoformat()
'P6DT0H50M3.010010012S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT1H0M10S'
>>> pd.Timedelta(days=500.5).isoformat()
'P500DT12H0M0S'
```

### pandas.Timedelta.round

**Timedelta.round(freq)**

Round the Timedelta to the specified resolution.

**Parameters**

- `freq` [str] Frequency string indicating the rounding resolution.

**Returns**

- a new Timedelta rounded to the given resolution of `freq`

**Raises**

- `ValueError` if the freq cannot be converted
pandas.Timedelta.to_numpy

Timedelta.to_numpy()

Convert the Timedelta to a NumPy timedelta64.

New in version 0.25.0.

This is an alias method for Timedelta.to_timedelta64(). The dtype and copy parameters are available here only for compatibility. Their values will not affect the return value.

Returns

numpy.timedelta64

See also:

Series.to_numpy Similar method for Series.

pandas.Timedelta.to_pytimedelta

Timedelta.to_pytimedelta()

Convert a pandas Timedelta object into a python datetime.timedelta object.

Timedelta objects are internally saved as numpy datetime64[ns] dtype. Use to_pytimedelta() to convert to object dtype.

Returns

datetime.timedelta or numpy.array of datetime.timedelta

See also:

to_timedelta Convert argument to Timedelta type.

Notes

Any nanosecond resolution will be lost.

pandas.Timedelta.to_timedelta64

Timedelta.to_timedelta64()

Return a numpy.timedelta64 object with ‘ns’ precision.

pandas.Timedelta.total_seconds

Timedelta.total_seconds()

Total seconds in the duration.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.Timedelta.view

Timedelta.view()

Array view compatibility.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timedelta.asm8</td>
<td>Return a numpy timedelta64 array scalar view.</td>
</tr>
<tr>
<td>Timedelta.components</td>
<td>Return a components namedtuple-like.</td>
</tr>
<tr>
<td>Timedelta.days</td>
<td>Number of days.</td>
</tr>
<tr>
<td>Timedelta.delta</td>
<td>Return the timedelta in nanoseconds (ns), for internal compatibility.</td>
</tr>
<tr>
<td>Timedelta.freq</td>
<td></td>
</tr>
<tr>
<td>Timedelta.is_populated</td>
<td></td>
</tr>
<tr>
<td>Timedelta.max</td>
<td></td>
</tr>
<tr>
<td>Timedelta.microseconds</td>
<td>Number of microseconds (&gt;= 0 and less than 1 second).</td>
</tr>
<tr>
<td>Timedelta.min</td>
<td></td>
</tr>
<tr>
<td>Timedelta.nanoseconds</td>
<td>Return the number of nanoseconds (n), where 0 &lt;= n &lt; 1 microsecond.</td>
</tr>
<tr>
<td>Timedelta.resolution</td>
<td></td>
</tr>
<tr>
<td>Timedelta.seconds</td>
<td>Number of seconds (&gt;= 0 and less than 1 day).</td>
</tr>
<tr>
<td>Timedelta.value</td>
<td></td>
</tr>
<tr>
<td>Timedelta.view</td>
<td>Array view compatibility.</td>
</tr>
</tbody>
</table>

pandas.Timedelta.freq

Timedelta.freq

pandas.Timedelta.is_populated

Timedelta.is_populated

pandas.Timedelta.max

Timedelta.max = Timedelta('106751 days 23:47:16.854775807')
pandas.Timedelta.min

Timedelta.min = Timedelta('-106752 days +00:12:43.145224193')

pandas.Timedelta.resolution

Timedelta.resolution = Timedelta('0 days 00:00:00.000000001')

pandas.Timedelta.value

Timedelta.value

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timedelta.ceil(freq)</td>
<td>Return a new Timedelta ceiled to this resolution.</td>
</tr>
<tr>
<td>Timedelta.floor(freq)</td>
<td>Return a new Timedelta floored to this resolution.</td>
</tr>
<tr>
<td>Timedelta.round(freq)</td>
<td>Round the Timedelta to the specified resolution.</td>
</tr>
<tr>
<td>Timedelta.to_pytimedelta</td>
<td>Convert a pandas Timedelta object into a python datetime.timedelta object.</td>
</tr>
<tr>
<td>Timedelta.to_timedelta64</td>
<td>Return a numpy.timedelta64 object with 'ns' precision.</td>
</tr>
<tr>
<td>Timedelta.to_numpy</td>
<td>Convert the Timedelta to a NumPy timedelta64.</td>
</tr>
<tr>
<td>Timedelta.total_seconds</td>
<td>Total seconds in the duration.</td>
</tr>
</tbody>
</table>

A collection of Timedelta may be stored in a TimedeltaArray.

arrays.TimedeltaArray(values[, dtype, freq, ...]) Pandas ExtensionArray for timedelta data.

pandas.arrays.TimedeltaArray

class pandas.arrays.TimedeltaArray(values, dtype='timedelta64[ns]', freq=None, copy=False)

Pandas ExtensionArray for timedelta data.

Warning: TimedeltaArray is currently experimental, and its API may change without warning. In particular, TimedeltaArray.dtype is expected to change to be an instance of an ExtensionDtype subclass.

Parameters
values [array-like] The timedelta data.
dtype [numpy.dtype] Currently, only numpy.dtype("timedelta64[ns]") is accepted.
freq [Offset, optional]copy [bool, default False] Whether to copy the underlying array of data.
3.5.4 Timespan data

pandas represents spans of times as `Period` objects.

3.5.5 Period

```python
Period([value, freq, ordinal, year, month, ...]) Represents a period of time.
```

`pandas.Period`

```python
class pandas.Period(value=None, freq=None, ordinal=None, year=None, month=None, quarter=None, day=None, hour=None, minute=None, second=None)
```

Represents a period of time.

**Parameters**

- `value` [Period or str, default None] The time period represented (e.g., ‘4Q2005’).
- `freq` [str, default None] One of pandas period strings or corresponding objects.
- `ordinal` [int, default None] The period offset from the proleptic Gregorian epoch.
- `year` [int, default None] Year value of the period.
- `month` [int, default 1] Month value of the period.
- `quarter` [int, default None] Quarter value of the period.
- `day` [int, default 1] Day value of the period.
- `hour` [int, default 0] Hour value of the period.
- `minute` [int, default 0] Minute value of the period.
- `second` [int, default 0] Second value of the period.
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>day</code></td>
<td>Get day of the month that a Period falls on.</td>
</tr>
<tr>
<td><code>day_of_week</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>day_of_year</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>dayofweek</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>dayofyear</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>days_in_month</code></td>
<td>Get the total number of days in the month that this period falls on.</td>
</tr>
<tr>
<td><code>daysinmonth</code></td>
<td>Get the total number of days of the month that the Period falls in.</td>
</tr>
<tr>
<td><code>end_time</code></td>
<td>Get the Timestamp for the end of the period.</td>
</tr>
<tr>
<td><code>freqstr</code></td>
<td>Return a string representation of the frequency.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>Get the hour of the day component of the Period.</td>
</tr>
<tr>
<td><code>is_leap_year</code></td>
<td>Return True if the period’s year is in a leap year.</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>Get minute of the hour component of the Period.</td>
</tr>
<tr>
<td><code>month</code></td>
<td>Return the month this Period falls on.</td>
</tr>
<tr>
<td><code>quarter</code></td>
<td>Return the quarter this Period falls on.</td>
</tr>
<tr>
<td><code>qyear</code></td>
<td>Fiscal year the Period lies in according to its starting-quarter.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>Get the second component of the Period.</td>
</tr>
<tr>
<td><code>start_time</code></td>
<td>Get the Timestamp for the start of the period.</td>
</tr>
<tr>
<td><code>week</code></td>
<td>Get the week of the year on the given Period.</td>
</tr>
<tr>
<td><code>weekday</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>weekofyear</code></td>
<td>Get the week of the year on the given Period.</td>
</tr>
<tr>
<td><code>year</code></td>
<td>Return the year this Period falls on.</td>
</tr>
</tbody>
</table>

### pandas.Period.day

Get day of the month that a Period falls on.

**Returns**

`int`

**See also:**

- `Period.dayofweek` Get the day of the week.
- `Period.dayofyear` Get the day of the year.
Examples

```python
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.day
11
```

pandas.Period.day_of_week

Period.day_of_week

Day of the week the period lies in, with Monday=0 and Sunday=6.

If the period frequency is lower than daily (e.g. hourly), and the period spans over multiple days, the day at the start of the period is used.

If the frequency is higher than daily (e.g. monthly), the last day of the period is used.

Returns

int Day of the week.

See also:

Period.day_of_week Day of the week the period lies in.
Period.weekday Alias of Period.day_of_week.
Period.day Day of the month.
Period.dayofyear Day of the year.

Examples

```python
>>> per = pd.Period('2017-12-31 22:00', 'H')
>>> per.day_of_week
6

For periods that span over multiple days, the day at the beginning of the period is returned.

```python
def test_he():
    per = pd.Period('2017-12-31 22:00', '4H')
    per.day_of_week
    >>> 6
    >>> per.start_time.day_of_week
    6

For periods with a frequency higher than days, the last day of the period is returned.

```python
def test_he():
    per = pd.Period('2018-01', 'M')
    per.day_of_week
    >>> 2
    >>> per.end_time.day_of_week
    2
```
pandas.Period.day_of_year

Period.day_of_year
Return the day of the year.

This attribute returns the day of the year on which the particular date occurs. The return value ranges between 1 to 365 for regular years and 1 to 366 for leap years.

Returns

int The day of year.

See also:

Period.day Return the day of the month.
Period.day_of_week Return the day of week.
PeriodIndex.day_of_year Return the day of year of all indexes.

Examples

>>> period = pd.Period("2015-10-23", freq='H')
>>> period.day_of_year
296
>>> period = pd.Period("2012-12-31", freq='D')
>>> period.day_of_year
366
>>> period = pd.Period("2013-01-01", freq='D')
>>> period.day_of_year
1

pandas.Period.dayofweek

Period.dayofweek
Day of the week the period lies in, with Monday=0 and Sunday=6.

If the period frequency is lower than daily (e.g. hourly), and the period spans over multiple days, the day at the start of the period is used.

If the frequency is higher than daily (e.g. monthly), the last day of the period is used.

Returns

int Day of the week.

See also:

Period.day_of_week Day of the week the period lies in.
Period.weekday Alias of Period.day_of_week.
Period.day Day of the month.
Period.dayofyear Day of the year.
Examples

```python
>>> per = pd.Period('2017-12-31 22:00', 'H')
>>> per.day_of_week
6
```

For periods that span over multiple days, the day at the beginning of the period is returned.

```python
>>> per = pd.Period('2017-12-31 22:00', '4H')
>>> per.day_of_week
6
>>> per.start_time.day_of_week
6
```

For periods with a frequency higher than days, the last day of the period is returned.

```python
>>> per = pd.Period('2018-01', 'M')
>>> per.day_of_week
2
>>> per.end_time.day_of_week
2
```

**pandas.Period.dayofyear**

`Period.dayofyear`

Return the day of the year.

This attribute returns the day of the year on which the particular date occurs. The return value ranges between 1 to 365 for regular years and 1 to 366 for leap years.

Returns

```
int   The day of year.
```

See also:

*`Period.day`  Return the day of the month.*

*`Period.day_of_week`  Return the day of week.*

*`PeriodIndex.day_of_year`  Return the day of year of all indexes.*

Examples

```python
>>> period = pd.Period("2015-10-23", freq='H')
>>> period.day_of_year
296
>>> period = pd.Period("2012-12-31", freq='D')
>>> period.day_of_year
366
>>> period = pd.Period("2013-01-01", freq='D')
>>> period.day_of_year
1
```
pandas.Period.days_in_month

Period.days_in_month
Get the total number of days in the month that this period falls on.

Returns
int

See also:

Period.daysinmonth  Gets the number of days in the month.
DatetimeIndex.daysinmonth  Gets the number of days in the month.
calendar.monthrange  Returns a tuple containing weekday (0-6 ~ Mon-Sun) and number of days (28-31).

Examples

```python
>>> p = pd.Period('2018-2-17')
>>> p.days_in_month
28

>>> pd.Period('2018-03-01').days_in_month
31
```

Handles the leap year case as well:

```python
>>> p = pd.Period('2016-2-17')
>>> p.days_in_month
29
```

pandas.Period.daysinmonth

Period.daysinmonth
Get the total number of days of the month that the Period falls in.

Returns
int

See also:

Period.days_in_month  Return the days of the month.
Period.dayofyear  Return the day of the year.
Examples

```python
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.daysinmonth
31
```

**pandas.Period.end_time**

Period.end_time

Get the Timestamp for the end of the period.

Returns

Timestamp

See also:

- Period.start_time  Return the start Timestamp.
- Period.dayofyear  Return the day of year.
- Period.daysinmonth  Return the days in that month.
- Period.dayofweek  Return the day of the week.

**pandas.Period.freqstr**

Period.freqstr

Return a string representation of the frequency.

**pandas.Period.hour**

Period.hour

Get the hour of the day component of the Period.

Returns

int  The hour as an integer, between 0 and 23.

See also:

- Period.second  Get the second component of the Period.
- Period.minute  Get the minute component of the Period.
**Examples**

```python
>>> p.hour
13
```

Period longer than a day

```python
>>> p = pd.Period("2018-03-11", freq="M")
>>> p.hour
0
```

**pandas.Period.is_leap_year**

Period.is_leap_year

Return True if the period’s year is in a leap year.

**pandas.Period.minute**

Period.minute

Get minute of the hour component of the Period.

Returns

int  The minute as an integer, between 0 and 59.

See also:

*Period.hour*  Get the hour component of the Period.
*Period.second*  Get the second component of the Period.

**Examples**

```python
>>> p.minute
3
```

**pandas.Period.month**

Period.month

Return the month this Period falls on.
**pandas.Period.quarter**

Period.quarter

Return the quarter this Period falls on.

**pandas.Period.qyear**

Period.qyear

Fiscal year the Period lies in according to its starting-quarter.

The year and the qyear of the period will be the same if the fiscal and calendar years are the same. When they are not, the fiscal year can be different from the calendar year of the period.

Returns

int The fiscal year of the period.

See also:

Period.year Return the calendar year of the period.

Examples

If the natural and fiscal year are the same, qyear and year will be the same.

```python
>>> per = pd.Period('2018Q1', freq='Q')
>>> per.qyear
2018
>>> per.year
2018
```

If the fiscal year starts in April (Q-MAR), the first quarter of 2018 will start in April 2017. year will then be 2018, but qyear will be the fiscal year, 2018.

```python
>>> per = pd.Period('2018Q1', freq='Q-MAR')
>>> per.start_time
Timestamp('2017-04-01 00:00:00')
>>> per.qyear
2018
>>> per.year
2017
```

**pandas.Period.second**

Period.second

Get the second component of the Period.

Returns

int The second of the Period (ranges from 0 to 59).

See also:

Period.hour Get the hour component of the Period.
**Period.minute** Get the minute component of the Period.

**Examples**

```python
>>> p.second
12
```

### pandas.Period.start_time

**Period.start_time**

Get the Timestamp for the start of the period.

**Returns**

Timestamp

See also:

- **Period.end_time** Return the end Timestamp.
- **Period.dayofyear** Return the day of year.
- **Period.daysinmonth** Return the days in that month.
- **Period.dayofweek** Return the day of the week.

**Examples**

```python
>>> period = pd.Period('2012-1-1', freq='D')
>>> period
Period('2012-01-01', 'D')

>>> period.start_time
Timestamp('2012-01-01 00:00:00')

>>> period.end_time
Timestamp('2012-01-01 23:59:59.999999999')
```

### pandas.Period.week

**Period.week**

Get the week of the year on the given Period.

**Returns**

int

See also:

- **Period.dayofweek** Get the day component of the Period.
- **Period.weekday** Get the day component of the Period.
Examples

```python
>>> p = pd.Period('2018-03-11', 'H')
>>> p.week
10

>>> p = pd.Period('2018-02-01', 'D')
>>> p.week
5

>>> p = pd.Period('2018-01-06', 'D')
>>> p.week
1
```

**pandas.Period.weekday**

Period.weekday

Day of the week the period lies in, with Monday=0 and Sunday=6.

If the period frequency is lower than daily (e.g. hourly), and the period spans over multiple days, the day at the start of the period is used.

If the frequency is higher than daily (e.g. monthly), the last day of the period is used.

Returns

int  Day of the week.

See also:

Period.dayofweek  Day of the week the period lies in.
Period.weekday  Alias of Period.dayofweek.
Period.day  Day of the month.
Period.dayofyear  Day of the year.

Examples

```python
>>> per = pd.Period('2017-12-31 22:00', 'H')
>>> per.dayofweek
6
```

For periods that span over multiple days, the day at the beginning of the period is returned.

```python
>>> per = pd.Period('2017-12-31 22:00', '4H')
>>> per.dayofweek
6

>>> per.start_time.dayofweek
6
```

For periods with a frequency higher than days, the last day of the period is returned.
```python
>>> per = pd.Period('2018-01', 'M')
>>> per.dayofweek
2
>>> per.end_time.dayofweek
2
```

### pandas.Period.weekofyear

**Period.weekofyear**

Get the week of the year on the given Period.

**Returns**

int

**See also:**

*Period.dayofweek* Get the day component of the Period.

*Period.weekday* Get the day component of the Period.

**Examples**

```python
>>> p = pd.Period("2018-03-11", "H")
>>> p.weekofyear
10
```  

```python
>>> p = pd.Period("2018-02-01", "D")
>>> p.weekofyear
5
```  

```python
>>> p = pd.Period("2018-01-06", "D")
>>> p.weekofyear
1
```  

### pandas.Period.year

**Period.year**

Return the year this Period falls on.

<table>
<thead>
<tr>
<th>freq</th>
<th>ordinal</th>
</tr>
</thead>
</table>

---

3.5. pandas arrays, scalars, and data types  2085
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asfreq</code></td>
<td>Convert Period to desired frequency, at the start or end of the interval.</td>
</tr>
<tr>
<td><code>now</code></td>
<td>Return the period of now's date.</td>
</tr>
<tr>
<td><code>strftime</code></td>
<td>Returns the string representation of the Period, depending on the selected fmt.</td>
</tr>
<tr>
<td><code>to_timestamp</code></td>
<td>Return the Timestamp representation of the Period.</td>
</tr>
</tbody>
</table>

**pandas.Period.asfreq**

`Period.asfreq()`

Convert Period to desired frequency, at the start or end of the interval.

**Parameters**

- **freq** [str] The desired frequency.
- **how** ['E', 'S', 'end', 'start'], default 'end' Start or end of the timespan.

**Returns**

- **resampled** [Period]

**pandas.Period.now**

`Period.now()`

Return the period of now's date.

**pandas.Period.strftime**

`Period.strftime()`

Returns the string representation of the Period, depending on the selected fmt. fmt must be a string containing one or several directives. The method recognizes the same directives as the time.strftime() function of the standard Python distribution, as well as the specific additional directives %f, %F, %q. (formatting & docs originally from scikits.timeries).
<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale’s abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale’s full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale’s abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale’s full month name.</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale’s appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%e</td>
<td>‘Fiscal’ year without a century as a decimal number [00,99]</td>
<td>(1)</td>
</tr>
<tr>
<td>%F</td>
<td>‘Fiscal’ year with a century as a decimal number</td>
<td>(2)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>(3)</td>
</tr>
<tr>
<td>%q</td>
<td>Quarter as a decimal number [01,04]</td>
<td></td>
</tr>
<tr>
<td>%s</td>
<td>Second as a decimal number [00,61].</td>
<td>(4)</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>(5)</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td>(5)</td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%z</td>
<td>Time zone name (no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal '%' character.</td>
<td></td>
</tr>
</tbody>
</table>

Notes

1. The %e directive is the same as %y if the frequency is not quarterly. Otherwise, it corresponds to the ‘fiscal’ year, as defined by the qyear attribute.
2. The %F directive is the same as %Y if the frequency is not quarterly. Otherwise, it corresponds to the ‘fiscal’ year, as defined by the qyear attribute.
3. The %p directive only affects the output hour field if the %I directive is used to parse the hour.
4. The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.
5. The %U and %W directives are only used in calculations when the day of the week and the year are specified.
Examples

```python
>>> a = Period(freq='Q-JUL', year=2006, quarter=1)
>>> a.strftime('%F-%q')
'2006-Q1'
>>> # Output the last month in the quarter of this date
>>> a.strftime('%b-%Y')
'Oct-2005'
>>> a = Period(freq='D', year=2001, month=1, day=1)
>>> a.strftime('%d-%b-%Y')
'01-Jan-2001'
>>> a.strftime('%b. %d, %Y was a %A')
'Jan. 01, 2001 was a Monday'
```

### pandas.Period.to_timestamp

Period.to_timestamp()  
Return the Timestamp representation of the Period.

Uses the target frequency specified at the part of the period specified by `how`, which is either `Start` or `Finish`.

**Parameters**

- `freq` [str or DateOffset] Target frequency. Default is ‘D’ if self.freq is week or longer and ‘S’ otherwise.

**Returns**

Timestamp
## Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Period.day</code></td>
<td>Get day of the month that a Period falls on.</td>
</tr>
<tr>
<td><code>Period.dayofweek</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>Period.day_of_week</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>Period.dayofyear</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>Period.day_of_year</code></td>
<td>Return the day of the year.</td>
</tr>
<tr>
<td><code>Period.days_in_month</code></td>
<td>Get the total number of days in the month that this period falls on.</td>
</tr>
<tr>
<td><code>Period.daysinmonth</code></td>
<td>Get the total number of days of the month that the Period falls in.</td>
</tr>
<tr>
<td><code>Period.end_time</code></td>
<td>Get the Timestamp for the end of the period.</td>
</tr>
<tr>
<td><code>Period.freq</code></td>
<td>Return a string representation of the frequency.</td>
</tr>
<tr>
<td><code>Period.freqstr</code></td>
<td>Get the hour of the day component of the Period.</td>
</tr>
<tr>
<td><code>Period.is_leap_year</code></td>
<td>Return True if the period’s year is in a leap year.</td>
</tr>
<tr>
<td><code>Period.minute</code></td>
<td>Get minute of the hour component of the Period.</td>
</tr>
<tr>
<td><code>Period.month</code></td>
<td>Return the month this Period falls on.</td>
</tr>
<tr>
<td><code>Period.ordinal</code></td>
<td>Return the quarter this Period falls on.</td>
</tr>
<tr>
<td><code>Period.qyear</code></td>
<td>Fiscal year the Period lies in according to its starting-quarter.</td>
</tr>
<tr>
<td><code>Period.second</code></td>
<td>Get the second component of the Period.</td>
</tr>
<tr>
<td><code>Period.start_time</code></td>
<td>Get the Timestamp for the start of the period.</td>
</tr>
<tr>
<td><code>Period.week</code></td>
<td>Get the week of the year on the given Period.</td>
</tr>
<tr>
<td><code>Period.weekday</code></td>
<td>Day of the week the period lies in, with Monday=0 and Sunday=6.</td>
</tr>
<tr>
<td><code>Period.weekofyear</code></td>
<td>Get the week of the year on the given Period.</td>
</tr>
<tr>
<td><code>Period.year</code></td>
<td>Return the year this Period falls on.</td>
</tr>
</tbody>
</table>

### pandas.Period.freq

Period.freq
pandas: powerful Python data analysis toolkit, Release 1.4.2

**Period.ordinal**

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period.asfreq</td>
<td>Convert Period to desired frequency, at the start or end of the interval.</td>
</tr>
<tr>
<td>Period.now</td>
<td>Return the period of now’s date.</td>
</tr>
<tr>
<td>Period.strftime</td>
<td>Returns the string representation of the Period, depending on the selected fmt.</td>
</tr>
<tr>
<td>Period.to_timestamp</td>
<td>Return the Timestamp representation of the Period.</td>
</tr>
</tbody>
</table>

A collection of *Period* may be stored in a `arrays.PeriodArray`. Every period in a `arrays.PeriodArray` must have the same *freq*.

```python
arrays.PeriodArray(values[, dtype, freq, copy])
```

**pandas.arrays.PeriodArray**

**class** `pandas.arrays.PeriodArray(values[, dtype, freq, copy])`

Pandas ExtensionArray for storing Period data.

Users should use `period_array()` to create new instances. Alternatively, `array()` can be used to create new instances from a sequence of Period scalars.

**Parameters**

- **values** [Union[PeriodArray, Series[period], ndarray[int], PeriodIndex]] The data to store. These should be arrays that can be directly converted to ordinals without inference or copy (PeriodArray, ndarray[int64]), or a box around such an array (Series[period], PeriodIndex).

- **dtype** [PeriodDtype, optional] A PeriodDtype instance from which to extract a freq. If both `freq` and `dtype` are specified, then the frequencies must match.

- **freq** [str or DateOffset] The freq to use for the array. Mostly applicable when `values` is an ndarray of integers, when `freq` is required. When `values` is a PeriodArray (or box around), it’s checked that `values.freq` matches `freq`.

- **copy** [bool, default False] Whether to copy the ordinals before storing.

**See also:**

- `Period` Represents a period of time.
- `PeriodIndex` Immutable Index for period data.
- `period_range` Create a fixed-frequency PeriodArray.
- `array` Construct a pandas array.
Notes

There are two components to a PeriodArray
  • ordinals : integer ndarray
  • freq : pd.tseries.offsets.Offset
The values are physically stored as a 1-D ndarray of integers. These are called “ordinals” and represent some kind of offset from a base.

The freq indicates the span covered by each element of the array. All elements in the PeriodArray have the same freq.

Attributes

None

Methods

None

PeriodDtype(freq)

An ExtensionDtype for Period data.

pandas.PeriodDtype

class pandas.PeriodDtype(freq=None)

An ExtensionDtype for Period data.

This is not an actual numpy dtype, but a duck type.

Parameters

freq [str or DateOffset] The frequency of this PeriodDtype.

Examples

>>> pd.PeriodDtype(freq='D')
period[D]

>>> pd.PeriodDtype(freq=pd.offsets.MonthEnd())
period[M]
Attributes

| freq | The frequency object of this PeriodDtype. |

```python
pandas.PeriodDtype.freq
```

**property** PeriodDtype.freq
The frequency object of this PeriodDtype.

Methods

None

3.5.6 Interval data

Arbitrary intervals can be represented as `Interval` objects.

```python
pandas.Interval
```

class pandas.Interval

Immutable object implementing an Interval, a bounded slice-like interval.

**Parameters**

- **left** [orderable scalar] Left bound for the interval.
- **right** [orderable scalar] Right bound for the interval.
- **closed** ['right', 'left', 'both', 'neither'], default 'right' Whether the interval is closed on the left-side, right-side, both or neither. See the Notes for more detailed explanation.

**See also:**

- `IntervalIndex` An Index of Interval objects that are all closed on the same side.
- `cut` Convert continuous data into discrete bins (Categorical of Interval objects).
- `qcut` Convert continuous data into bins (Categorical of Interval objects) based on quantiles.
- `Period` Represents a period of time.

**Notes**

The parameters `left` and `right` must be from the same type, you must be able to compare them and they must satisfy `left <= right`.

A closed interval (in mathematics denoted by square brackets) contains its endpoints, i.e. the closed interval `[0, 5]` is characterized by the conditions `0 <= x <= 5`. This is what `closed='both'` stands for. An open interval (in mathematics denoted by parentheses) does not contain its endpoints, i.e. the open interval `(0, 5)` is characterized by the conditions `0 < x < 5`. This is what `closed='neither'` stands for. Intervals can also be half-open or half-closed, i.e. `[0, 5)` is described by `0 <= x < 5 (closed='left')` and `(0, 5]` is described by `0 < x <= 5 (closed='right').`
Examples

It is possible to build Intervals of different types, like numeric ones:

```python
>>> iv = pd.Interval(left=0, right=5)
>>> iv
Interval(0, 5, closed='right')
```

You can check if an element belongs to it

```python
>>> 2.5 in iv
True
```

You can test the bounds (closed='right', so 0 < x <= 5):

```python
>>> 0 in iv
False
>>> 5 in iv
True
>>> 0.0001 in iv
True
```

Calculate its length

```python
>>> iv.length
5
```

You can operate with + and * over an Interval and the operation is applied to each of its bounds, so the result depends on the type of the bound elements

```python
>>> shifted_iv = iv + 3
>>> shifted_iv
Interval(3, 8, closed='right')
>>> extended_iv = iv * 10.0
>>> extended_iv
Interval(0.0, 50.0, closed='right')
```

To create a time interval you can use Timestamps as the bounds

```python
>>> year_2017 = pd.Interval(pd.Timestamp('2017-01-01 00:00:00'),
... pd.Timestamp('2018-01-01 00:00:00'),
... closed='left')
>>> pd.Timestamp('2017-01-01 00:00') in year_2017
True
>>> year_2017.length
Timedelta('365 days 00:00:00')
```
Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closed</td>
<td>Whether the interval is closed on the left-side, right-side, both or neither.</td>
</tr>
<tr>
<td>closed_left</td>
<td>Check if the interval is closed on the left side.</td>
</tr>
<tr>
<td>closed_right</td>
<td>Check if the interval is closed on the right side.</td>
</tr>
<tr>
<td>is_empty</td>
<td>Indicates if an interval is empty, meaning it contains no points.</td>
</tr>
<tr>
<td>left</td>
<td>Left bound for the interval.</td>
</tr>
<tr>
<td>length</td>
<td>Return the length of the Interval.</td>
</tr>
<tr>
<td>mid</td>
<td>Return the midpoint of the Interval.</td>
</tr>
<tr>
<td>open_left</td>
<td>Check if the interval is open on the left side.</td>
</tr>
<tr>
<td>open_right</td>
<td>Check if the interval is open on the right side.</td>
</tr>
<tr>
<td>right</td>
<td>Right bound for the interval.</td>
</tr>
</tbody>
</table>

**pandas.Interval.closed**

Interval.closed

Whether the interval is closed on the left-side, right-side, both or neither.

**pandas.Interval.closed_left**

Interval.closed_left

Check if the interval is closed on the left side.

For the meaning of closed and open see Interval.

Returns

bool True if the Interval is closed on the left-side.

**pandas.Interval.closed_right**

Interval.closed_right

Check if the interval is closed on the right side.

For the meaning of closed and open see Interval.

Returns

bool True if the Interval is closed on the left-side.
pandas.Interval.is_empty

Interval.is_empty
Indicates if an interval is empty, meaning it contains no points.
New in version 0.25.0.

Returns

bool or ndarray A boolean indicating if a scalar Interval is empty, or a boolean ndarray positionally indicating if an Interval in an IntervalArray or IntervalIndex is empty.

Examples

An Interval that contains points is not empty:

>>> pd.Interval(0, 1, closed='right').is_empty
False

An Interval that does not contain any points is empty:

>>> pd.Interval(0, 0, closed='right').is_empty
True
>>> pd.Interval(0, 0, closed='left').is_empty
True
>>> pd.Interval(0, 0, closed='neither').is_empty
True

An Interval that contains a single point is not empty:

>>> pd.Interval(0, 0, closed='both').is_empty
False

An IntervalArray or IntervalIndex returns a boolean ndarray positionally indicating if an Interval is empty:

>>> ivs = [pd.Interval(0, 0, closed='neither'),
...        pd.Interval(1, 2, closed='neither')]
>>> pd.arrays.IntervalArray(ivs).is_empty
array([ True, False])

Missing values are not considered empty:

>>> ivs = [pd.Interval(0, 0, closed='neither'), np.nan]
>>> pd.IntervalIndex(ivs).is_empty
array([[ True, False]])
pandas.Interval.left

Interval.left
   Left bound for the interval.

pandas.Interval.length

Interval.length
   Return the length of the Interval.

pandas.Interval.mid

Interval.mid
   Return the midpoint of the Interval.

pandas.Interval.open_left

Interval.open_left
   Check if the interval is open on the left side.
   For the meaning of closed and open see Interval.
   Returns
   bool True if the Interval is closed on the left-side.

pandas.Interval.open_right

Interval.open_right
   Check if the interval is open on the right side.
   For the meaning of closed and open see Interval.
   Returns
   bool True if the Interval is closed on the left-side.

pandas.Interval.right

Interval.right
   Right bound for the interval.
Methods

overlaps

pandas.Interval.overlaps

Interval.overlaps()

Check whether two Interval objects overlap.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

Parameters

other [Interval] Interval to check against for an overlap.

Returns

bool True if the two intervals overlap.

See also:

IntervalArray.overlaps The corresponding method for IntervalArray.

IntervalIndex.overlaps The corresponding method for IntervalIndex.

Examples

```python
>>> i1 = pd.Interval(0, 2)
>>> i2 = pd.Interval(1, 3)
>>> i1.overlaps(i2)
True
>>> i3 = pd.Interval(4, 5)
>>> i1.overlaps(i3)
False
```

Intervals that share closed endpoints overlap:

```python
>>> i4 = pd.Interval(0, 1, closed='both')
>>> i5 = pd.Interval(1, 2, closed='both')
>>> i4.overlaps(i5)
True
```

Intervals that only have an open endpoint in common do not overlap:

```python
>>> i6 = pd.Interval(1, 2, closed='neither')
>>> i4.overlaps(i6)
False
```
## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Interval.closed</code></td>
<td>Whether the interval is closed on the left-side, right-side, both or neither.</td>
</tr>
<tr>
<td><code>Interval.closed_left</code></td>
<td>Check if the interval is closed on the left side.</td>
</tr>
<tr>
<td><code>Interval.closed_right</code></td>
<td>Check if the interval is closed on the right side.</td>
</tr>
<tr>
<td><code>Interval.is_empty</code></td>
<td>Indicates if an interval is empty, meaning it contains no points.</td>
</tr>
<tr>
<td><code>Interval.left</code></td>
<td>Left bound for the interval.</td>
</tr>
<tr>
<td><code>Interval.length</code></td>
<td>Return the length of the Interval.</td>
</tr>
<tr>
<td><code>Interval.mid</code></td>
<td>Return the midpoint of the Interval.</td>
</tr>
<tr>
<td><code>Interval.open_left</code></td>
<td>Check if the interval is open on the left side.</td>
</tr>
<tr>
<td><code>Interval.open_right</code></td>
<td>Check if the interval is open on the right side.</td>
</tr>
<tr>
<td><code>Interval.overlaps</code></td>
<td>Check whether two Interval objects overlap.</td>
</tr>
<tr>
<td><code>Interval.right</code></td>
<td>Right bound for the interval.</td>
</tr>
</tbody>
</table>

A collection of intervals may be stored in an `arrays.IntervalArray`.

```python
arrays.IntervalArray(data[, closed, dtype, ...])
```

Pandas array for interval data that are closed on the same side.

### pandas.arrays.IntervalArray

**class** `pandas.arrays.IntervalArray`(data, closed=None, dtype=None, copy=False, verify_integrity=True)

Pandas array for interval data that are closed on the same side.

New in version 0.24.0.

**Parameters**

- **data** [array-like (1-dimensional)] Array-like containing Interval objects from which to build the IntervalArray.
- **closed** [‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’ Whether the intervals are closed on the left-side, right-side, both or neither.
- **dtype** [dtype or None, default None] If None, dtype will be inferred.
- **copy** [bool, default False] Copy the input data.
- **verify_integrity** [bool, default True] Verify that the IntervalArray is valid.

**See also:**

- **Index** The base pandas Index type.
- **Interval** A bounded slice-like interval; the elements of an IntervalArray.
- **interval_range** Function to create a fixed frequency IntervalIndex.
- **cut** Bin values into discrete Intervals.
- **qcut** Bin values into equal-sized Intervals based on rank or sample quantiles.
Notes

See the user guide for more.

Examples

A new `IntervalArray` can be constructed directly from an array-like of `Interval` objects:

```python
>>> pd.arrays.IntervalArray([pd.Interval(0, 1), pd.Interval(1, 5)])
<IntervalArray>
  [(0, 1], (1, 5]]
Length: 2, dtype: interval[int64, right]
```

It may also be constructed using one of the constructor methods: `IntervalArray.from_arrays()`, `IntervalArray.from_breaks()`, and `IntervalArray.from_tuples()`.

Attributes

- `left` property of `IntervalArray` returns the left endpoints of each Interval in the `IntervalArray` as an Index.
- `right` property of `IntervalArray` returns the right endpoints of each Interval in the `IntervalArray` as an Index.
- `closed` property returns whether the intervals are closed on the left-side, right-side, both or neither.
- `mid` property returns the midpoint of each Interval in the `IntervalArray` as an Index.
- `length` property returns an Index with entries denoting the length of each Interval in the `IntervalArray`.
- `is_empty` property indicates if an interval is empty, meaning it contains no points.
- `is_non_overlapping_monotonic` property returns True if the `IntervalArray` is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False.

```
pandas.arrays.IntervalArray.left

property IntervalArray.left
    Return the left endpoints of each Interval in the IntervalArray as an Index.

pandas.arrays.IntervalArray.right

property IntervalArray.right
    Return the right endpoints of each Interval in the IntervalArray as an Index.
```
pandas.arrays.IntervalArray.closed

**property** IntervalArray.closed

Whether the intervals are closed on the left-side, right-side, both or neither.

pandas.arrays.IntervalArray.mid

**property** IntervalArray.mid

Return the midpoint of each Interval in the IntervalArray as an Index.

pandas.arrays.IntervalArray.length

**property** IntervalArray.length

Return an Index with entries denoting the length of each Interval in the IntervalArray.

pandas.arrays.IntervalArray.is_empty

IntervalArray.is_empty

Indicates if an interval is empty, meaning it contains no points.

New in version 0.25.0.

**Returns**

bool or ndarray A boolean indicating if a scalar Interval is empty, or a boolean ndarray positionally indicating if an Interval in an IntervalArray or IntervalIndex is empty.

**Examples**

An Interval that contains points is not empty:

```python
>>> pd.Interval(0, 1, closed='right').is_empty
False
```

An Interval that does not contain any points is empty:

```python
>>> pd.Interval(0, 0, closed='right').is_empty
True
>>> pd.Interval(0, 0, closed='left').is_empty
True
>>> pd.Interval(0, 0, closed='neither').is_empty
True
```

An Interval that contains a single point is not empty:

```python
>>> pd.Interval(0, 0, closed='both').is_empty
False
```

An IntervalArray or IntervalIndex returns a boolean ndarray positionally indicating if an Interval is empty:
```python
>>> ivs = [pd.Interval(0, 0, closed='neither'), pd.Interval(1, 2, closed='neither')]
>>> pd.arrays.IntervalArray(ivs).is_empty
array([True, False])

Missing values are not considered empty:
```n
```python
>>> ivs = [pd.Interval(0, 0, closed='neither'), np.nan]
>>> pd.IntervalIndex(ivs).is_empty
array([True, False])
```

### pandas.arrays.IntervalArray.is_non_overlapping_monotonic

**property IntervalArray.is_non_overlapping_monotonic**

Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False.

### Methods

**from_arrays**

```python
from_arrays(self, left, right, closed='right', copy=False, dtype=None)
```

Construct from two arrays defining the left and right bounds.

**Parameters**

- `left` [array-like (1-dimensional)] Left bounds for each interval.
- `right` [array-like (1-dimensional)] Right bounds for each interval.
- `closed` [‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’ Whether the intervals are closed on the left-side, right-side, both or neither.
- `copy` [bool, default False] Copy the data.
- `dtype` [dtype, optional] If None, dtype will be inferred.

**Returns**

- IntervalArray

### pandas.arrays.IntervalArray.from_arrays

**classmethod IntervalArray.from_arrays**

Construct from two arrays defining the left and right bounds.

**Parameters**

- `left` [array-like (1-dimensional)] Left bounds for each interval.
- `right` [array-like (1-dimensional)] Right bounds for each interval.
- `closed` [‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’ Whether the intervals are closed on the left-side, right-side, both or neither.
- `copy` [bool, default False] Copy the data.
- `dtype` [dtype, optional] If None, dtype will be inferred.

**Returns**

- IntervalArray
Raises

ValueError When a value is missing in only one of left or right. When a value in left is greater than the corresponding value in right.

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_breaks Construct an IntervalArray from an array of splits.

IntervalArray.from_tuples Construct an IntervalArray from an array-like of tuples.

Notes

Each element of left must be less than or equal to the right element at the same position. If an element is missing, it must be missing in both left and right. A TypeError is raised when using an unsupported type for left or right. At the moment, ‘category’, ‘object’, and ‘string’ subtypes are not supported.

```python
>>> pd.arrays.IntervalArray.from_arrays([0, 1, 2], [1, 2, 3])
<IntervalArray>
[(0, 1], (1, 2], (2, 3]]
Length: 3, dtype: interval[int64, right]
```

pandas.arrays.IntervalArray.from_tuples

classmethod IntervalArray.from_tuples(data, closed='right', copy=False, dtype=None)

Construct an IntervalArray from an array-like of tuples.

Parameters

data [array-like (1-dimensional)] Array of tuples.

closed [[‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

IntervalArray

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_arrays Construct an IntervalArray from a left and right array.

IntervalArray.from_breaks Construct an IntervalArray from an array of splits.
### pandas: powerful Python data analysis toolkit, Release 1.4.2

#### Examples

```python
>>> pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 2)])
<IntervalArray>
[(0, 1], (1, 2]
Length: 2, dtype: interval[int64, right]
```

#### pandas.arrays.IntervalArray.from_breaks

**classmethod**  
`IntervalArray.from_breaks(breaks, closed='right', copy=False, dtype=None)`  
Construct an IntervalArray from an array of splits.

**Parameters**

- `breaks` [array-like (1-dimensional)] Left and right bounds for each interval.
- `closed` [‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’] Whether the intervals are closed on the left-side, right-side, both or neither.
- `copy` [bool, default False] Copy the data.
- `dtype` [dtype or None, default None] If None, dtype will be inferred.

**Returns**

`IntervalArray`

**See also:**

- `interval_range` Function to create a fixed frequency IntervalIndex.
- `IntervalArray.from_arrays` Construct from a left and right array.
- `IntervalArray.from_tuples` Construct from a sequence of tuples.

#### Examples

```python
>>> pd.arrays.IntervalArray.from_breaks([0, 1, 2, 3])
<IntervalArray>
[(0, 1], (1, 2], (2, 3]
Length: 3, dtype: interval[int64, right]
```

#### pandas.arrays.IntervalArray.contains

`IntervalArray.contains(other)`  
Check elementwise if the Intervals contain the value.

Return a boolean mask whether the value is contained in the Intervals of the IntervalArray.

New in version 0.25.0.

**Parameters**

- `other` [scalar] The value to check whether it is contained in the Intervals.

**Returns**

3.5. pandas arrays, scalars, and data types 2103
boolean array

See also:

**Interval.contains**  Check whether Interval object contains value.

**IntervalArray.overlaps**  Check if an Interval overlaps the values in the IntervalArray.

**Examples**

```python
>>> intervals = pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 3), (2, 4)])
>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]
Length: 3, dtype: interval[int64, right]

>>> intervals.contains(0.5)
array([ True, False, False])
```

**pandas.arrays.IntervalArray.overlaps**

IntervalArray.overlaps\((other)\)

Check elementwise if an Interval overlaps the values in the IntervalArray.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

**Parameters**

- **other**  [IntervalArray] Interval to check against for an overlap.

**Returns**

ndarray  Boolean array positionally indicating where an overlap occurs.

See also:

**Interval.overlaps**  Check whether two Interval objects overlap.

**Examples**

```python
>>> data = [(0, 1), (1, 3), (2, 4)]

>>> intervals = pd.arrays.IntervalArray.from_tuples(data)

>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]
Length: 3, dtype: interval[int64, right]

>>> intervals.overlaps(pd.Interval(0.5, 1.5))
array([ True, True, False])
```

Intervals that share closed endpoints overlap:
intervals.overlaps(pd.Interval(1, 3, closed='left'))
array([ True,  True,  True])

Intervals that only have an open endpoint in common do not overlap:

intervals.overlaps(pd.Interval(1, 2, closed='right'))
array([False,  True, False])

pandas.arrays.IntervalArray.set_closed

IntervalArray.set_closed(closed)

Return an IntervalArray identical to the current one, but closed on the specified side.

Parameters

closed [{'left', 'right', 'both', 'neither'}] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

new_index [IntervalArray]

Examples

index = pd.arrays.IntervalArray.from_breaks(range(4))

index
<IntervalArray>
[ (0, 1], (1, 2], (2, 3] ]
Length: 3, dtype: interval[int64, right]

index.set_closed('both')
<IntervalArray>
[ [0, 1], [1, 2], [2, 3] ]
Length: 3, dtype: interval[int64, both]

pandas.arrays.IntervalArray.to_tuples

IntervalArray.to_tuples(na_tuple=True)

Return an ndarray of tuples of the form (left, right).

Parameters

na_tuple [bool, default True] Returns NA as a tuple if True, (nan, nan), or just as the NA value itself if False, nan.

Returns

tuples: ndarray

IntervalDtype([subtype, closed]) An ExtensionDtype for Interval data.
pandas.IntervalDtype

class pandas.IntervalDtype(subtype=None, closed=None)
    An ExtensionDtype for Interval data.
    This is not an actual numpy dtype, but a duck type.
    Parameters

Examples

>>> pd.IntervalDtype(subtype='int64', closed='both')
interval[int64, both]

Attributes

subtype
    The dtype of the Interval bounds.

pandas.IntervalDtype.subtype

property IntervalDtype.subtype
    The dtype of the Interval bounds.

Methods

None

3.5.7 Nullable integer

numpy.ndarray cannot natively represent integer-data with missing values. pandas provides this through arrays.IntegerArray.

arrays.IntegerArray(values, mask[, copy])
    Array of integer (optional missing) values.

pandas.arrays.IntegerArray

class pandas.arrays=IntegerArray(values, mask[, copy])
    Array of integer (optional missing) values.
    Changed in version 1.0.0: Now uses pandas.NA as the missing value rather than numpy.nan.

Warning: IntegerArray is currently experimental, and its API or internal implementation may change without warning.
We represent an IntegerArray with 2 numpy arrays:

- data: contains a numpy integer array of the appropriate dtype
- mask: a boolean array holding a mask on the data, True is missing

To construct an IntegerArray from generic array-like input, use `pandas.array()` with one of the integer dtypes (see examples).

See [Nullable integer data type](#) for more.

**Parameters**

- **mask** [numpy.ndarray] A 1-d boolean-dtype array indicating missing values.
- **copy** [bool, default False] Whether to copy the values and mask.

**Returns**

- IntegerArray

**Examples**

Create an IntegerArray with `pandas.array()`.

```python
>>> int_array = pd.array([1, None, 3], dtype=pd.Int32Dtype())
>>> int_array
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: Int32
```

String aliases for the dtypes are also available. They are capitalized.

```python
>>> pd.array([1, None, 3], dtype='Int32')
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: Int32
```

```python
>>> pd.array([1, None, 3], dtype='UInt16')
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: UInt16
```

**Attributes**

- None
Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Dtype()</td>
<td>An ExtensionDtype for int8 integer data.</td>
</tr>
<tr>
<td>Int16Dtype()</td>
<td>An ExtensionDtype for int16 integer data.</td>
</tr>
<tr>
<td>Int32Dtype()</td>
<td>An ExtensionDtype for int32 integer data.</td>
</tr>
<tr>
<td>Int64Dtype()</td>
<td>An ExtensionDtype for int64 integer data.</td>
</tr>
<tr>
<td>UInt8Dtype()</td>
<td>An ExtensionDtype for uint8 integer data.</td>
</tr>
<tr>
<td>UInt16Dtype()</td>
<td>An ExtensionDtype for uint16 integer data.</td>
</tr>
<tr>
<td>UInt32Dtype()</td>
<td>An ExtensionDtype for uint32 integer data.</td>
</tr>
<tr>
<td>UInt64Dtype()</td>
<td>An ExtensionDtype for uint64 integer data.</td>
</tr>
</tbody>
</table>

`pandas.Int8Dtype`

class pandas.Int8Dtype

An ExtensionDtype for int8 integer data.

Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

`pandas.Int16Dtype`

class pandas.Int16Dtype

An ExtensionDtype for int16 integer data.

Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

Methods

None

pandas.Int32Dtype
class pandas.Int32Dtype
   An ExtensionDtype for int32 integer data.
   Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

None

Methods

None

pandas.Int64Dtype
class pandas.Int64Dtype
   An ExtensionDtype for int64 integer data.
   Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

None

Methods

None

pandas.UInt8Dtype
class pandas.UInt8Dtype
   An ExtensionDtype for uint8 integer data.
   Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.
pandas: powerful Python data analysis toolkit, Release 1.4.2

Attributes

None

Methods

None

**pandas.UInt16Dtype**

class pandas.UInt16Dtype

An ExtensionDtype for uint16 integer data.

Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

None

Methods

None

**pandas.UInt32Dtype**

class pandas.UInt32Dtype

An ExtensionDtype for uint32 integer data.

Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

None

Methods

None
pandas.UInt64Dtype

class pandas.UInt64Dtype
   An ExtensionDtype for uint64 integer data.
   Changed in version 1.0.0: Now uses pandas.NA as its missing value, rather than numpy.nan.

Attributes

None

Methods

None

3.5.8 Categorical data

pandas defines a custom data type for representing data that can take only a limited, fixed set of values. The dtype of a Categorical can be described by a CategoricalDtype.

CategoricalDtype([categories, ordered]) Type for categorical data with the categories and orderedness.

pandas.CategoricalDtype

class pandas.CategoricalDtype(categories=None, ordered=False)
   Type for categorical data with the categories and orderedness.
   Parameters
      categories [sequence, optional] Must be unique, and must not contain any nulls. The categories are stored in an Index, and if an index is provided the dtype of that index will be used.
      ordered [bool or None, default False] Whether or not this categorical is treated as a ordered categorical. None can be used to maintain the ordered value of existing categoricals when used in operations that combine categoricals, e.g. astype, and will resolve to False if there is no existing ordered to maintain.
   See also:
      Categorical Represent a categorical variable in classic R / S-plus fashion.
Notes

This class is useful for specifying the type of a Categorical independent of the values. See CategoricalDtype for more.

Examples

```python
>>> t = pd.CategoricalDtype(categories=['b', 'a'], ordered=True)
>>> pd.Series(['a', 'b', 'a', 'c'], dtype=t)
0    a
1    b
2    a
3   NaN
dtype: category
Categories (2, object): ['b' < 'a']
```

An empty CategoricalDtype with a specific dtype can be created by providing an empty index. As follows,

```python
>>> pd.CategoricalDtype(pd.DatetimeIndex([])).categories.dtype
dtype('<M8[ns]')
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories</td>
<td>An Index containing the unique categories allowed.</td>
</tr>
<tr>
<td>ordered</td>
<td>Whether the categories have an ordered relationship.</td>
</tr>
</tbody>
</table>

```
pandas.CategoricalDtype.categories

property CategoricalDtype.categories

    An Index containing the unique categories allowed.

```

```
pandas.CategoricalDtype.ordered

property CategoricalDtype.ordered

    Whether the categories have an ordered relationship.

```

Methods

None

Categorical data can be stored in a `pandas.Categorical`
Represent a categorical variable in classic R / S-plus fashion.

**pandas.Categorical**

class pandas.Categorical(values, categories=None, ordered=None, dtype=None, fastpath=False, copy=True)

Represent a categorical variable in classic R / S-plus fashion.

Categoricals can only take on only a limited, and usually fixed, number of possible values (categories). In contrast to statistical categorical variables, a Categorical might have an order, but numerical operations (additions, divisions, ...) are not possible.

All values of the Categorical are either in categories or np.nan. Assigning values outside of categories will raise a ValueError. Order is defined by the order of the categories, not lexical order of the values.

**Parameters**

values [list-like] The values of the categorical. If categories are given, values not in categories will be replaced with NaN.

categories [Index-like (unique), optional] The unique categories for this categorical. If not given, the categories are assumed to be the unique values of values (sorted, if possible, otherwise in the order in which they appear).

ordered [bool, default False] Whether or not this categorical is treated as a ordered categorical. If True, the resulting categorical will be ordered. An ordered categorical respects, when sorted, the order of its categories attribute (which in turn is the categories argument, if provided).

dtype [CategoricalDtype] An instance of CategoricalDtype to use for this categorical.

**Raises**

ValueError If the categories do not validate.

TypeError If an explicit ordered=True is given but no categories and the values are not sortable.

**See also:**

CategoricalDtype Type for categorical data.

CategoricalIndex An Index with an underlying Categorical.

**Notes**

See the user guide for more.

**Examples**

```python
>>> pd.Categorical([1, 2, 3, 1, 2, 3])
[1, 2, 3, 1, 2, 3]
Categories (3, int64): [1, 2, 3]
```

```python
>>> pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```
Missing values are not included as a category.

```python
>>> c = pd.Categorical([1, 2, 3, 1, 2, 3, np.nan])
>>> c
[1, 2, 3, 1, 2, 3, NaN]
Categories (3, int64): [1, 2, 3]
```

However, their presence is indicated in the `codes` attribute by code `-1`.

```python
>>> c.codes
array([ 0, 1, 2, 0, 1, 2, -1], dtype=int8)
```

Ordered `Categoricals` can be sorted according to the custom order of the categories and can have a min and max value.

```python
>>> c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'], ordered=True,
...                     categories=['c', 'b', 'a'])
>>> c
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['c' < 'b' < 'a']
```

### Attributes

- **categories**
  The categories of this categorical.
- **codes**
  The category codes of this categorical.
- **ordered**
  Whether the categories have an ordered relationship.
- **dtype**
  The `CategoricalDtype` for this instance.

```python
pandas.Categorical.categories
```

**property** `Categorical.categories`

The categories of this categorical.

- Setting assigns new values to each category (effectively a rename of each individual category).
- The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.
- Assigning to `categories` is a inplace operation!

**Raises**

- `ValueError` If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories.

**See also:**

- `rename_categories` Rename categories.
- `reorder_categories` Reorder categories.
- `add_categories` Add new categories.
- `remove_categories` Remove the specified categories.
**remove_unused_categories**  Remove categories which are not used.

**set_categories**  Set the categories to the specified ones.

### pandas.Categorical.codes

**property**  Categorical.codes

The category codes of this categorical.

Codes are an array of integers which are the positions of the actual values in the categories array.

There is no setter, use the other categorical methods and the normal item setter to change values in the categorical.

**Returns**

ndarray[int]  A non-writable view of the codes array.

### pandas.Categorical.ordered

**property**  Categorical.ordered

Whether the categories have an ordered relationship.

### pandas.Categorical.dtype

**property**  Categorical.dtype

The CategoricalDtype for this instance.

### Methods

<table>
<thead>
<tr>
<th>from_codes(codes[, categories, ordered, dtype])</th>
<th>Make a Categorical type from codes and categories or dtype.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>array</strong>([dtype])</td>
<td>The numpy array interface.</td>
</tr>
</tbody>
</table>

### pandas.Categorical.from_codes

**classmethod**  Categorical.from_codes(codes, categories=None, ordered=None, dtype=None)

Make a Categorical type from codes and categories or dtype.

This constructor is useful if you already have codes and categories/dtype and so do not need the (computation intensive) factorization step, which is usually done on the constructor.

If your data does not follow this convention, please use the normal constructor.

**Parameters**

- **codes**  [array-like of int]  An integer array, where each integer points to a category in categories or dtype.categories, or else is -1 for NaN.

- **categories**  [index-like, optional]  The categories for the categorical. Items need to be unique. If the categories are not given here, then they must be provided in dtype.
ordered [bool, optional] Whether or not this categorical is treated as an ordered categorical. If not given here or in dtype, the resulting categorical will be unordered.

dtype [CategoricalDtype or “category”, optional] If CategoricalDtype, cannot be used together with categories or ordered.

Returns

Categorical

Examples

```python
dtype = pd.CategoricalDtype(["a", "b"], ordered=True)
pd.Categorical.from_codes(codes=[0, 1, 0, 1], dtype=dtype)
['a', 'b', 'a', 'b']
Categories (2, object): ['a' < 'b']
```

pandas.Categorical.__array__

Categorical.__array__(dtype=None)
The numpy array interface.

Returns

numpy.array A numpy array of either the specified dtype or, if dtype==None (default), the same dtype as categorical.categories.dtype.

The alternative Categorical.from_codes() constructor can be used when you have the categories and integer codes already:

```python
Categorical.from_codes(codes[, categories, ...]) Make a Categorical type from codes and categories or dtype.
```

The dtype information is available on the Categorical

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categorical.dtype</td>
<td>The CategoricalDtype for this instance.</td>
</tr>
<tr>
<td>Categorical.categories</td>
<td>The categories of this categorical.</td>
</tr>
<tr>
<td>Categorical.ordered</td>
<td>Whether the categories have an ordered relationship.</td>
</tr>
<tr>
<td>Categorical.codes</td>
<td>The category codes of this categorical.</td>
</tr>
</tbody>
</table>

np.asarray(categorical) works by implementing the array interface. Be aware, that this converts the Categorical back to a NumPy array, so categories and order information is not preserved!

Categorical.__array__(dtype) The numpy array interface.

A Categorical can be stored in a Series or DataFrame. To create a Series of dtype category, use cat = s. astype(dtype) or Series(..., dtype=dtype) where dtype is either

- the string 'category'
- an instance of CategoricalDtype.

If the Series is of dtype CategoricalDtype, Series.cat can be used to change the categorical data. See Categorical accessor for more.
### 3.5.9 Sparse data

Data where a single value is repeated many times (e.g. 0 or NaN) may be stored efficiently as a `arrays.SparseArray`.

`arrays.SparseArray(data[, sparse_index, ...])` An ExtensionArray for storing sparse data.

#### pandas.arrays.SparseArray

class pandas.arrays.SparseArray(data, sparse_index=None, index=None, fill_value=None, kind='integer', dtype=None, copy=False)

An ExtensionArray for storing sparse data.

**Parameters**

- **data** [array-like or scalar] A dense array of values to store in the SparseArray. This may contain `fill_value`.
- **sparse_index** [SparseIndex, optional]
- **index** [Index] Deprecated since version 1.4.0: Use a function like `np.full` to construct an array with the desired repeats of the scalar value instead.
- **fill_value** [scalar, optional] Elements in data that are `fill_value` are not stored in the SparseArray. For memory savings, this should be the most common value in `data`. By default, `fill_value` depends on the dtype of `data`:

<table>
<thead>
<tr>
<th><code>data.dtype</code></th>
<th><code>na_value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td><code>np.nan</code></td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>bool</td>
<td><code>False</code></td>
</tr>
<tr>
<td><code>datetime64</code></td>
<td><code>pd.NaT</code></td>
</tr>
<tr>
<td><code>timedelta64</code></td>
<td><code>pd.NaT</code></td>
</tr>
</tbody>
</table>

The fill value is potentially specified in three ways. In order of precedence, these are

1. The `fill_value` argument
2. `dtype.fill_value` if `fill_value` is None and `dtype` is a `SparseDtype`
3. `data.dtype.fill_value` if `fill_value` is None and `dtype` is not a `SparseDtype` and `data` is a `SparseArray`.

- **kind** [str] Can be ‘integer’ or ‘block’, default is ‘integer’. The type of storage for sparse locations.
  - ‘block’: Stores a block and `block_length` for each contiguous span of sparse values. This is best when sparse data tends to be clumped together, with large regions of fill-value values between sparse values.
  - ‘integer’: uses an integer to store the location of each sparse value.
- **dtype** [np.dtype or SparseDtype, optional] The dtype to use for the SparseArray. For numpy dtypes, this determines the dtype of `self.sp_values`. For SparseDtype, this determines `self.sp_values` and `self.fill_value`.
- **copy** [bool, default False] Whether to explicitly copy the incoming `data` array.
Examples

```python
>>> from pandas.arrays import SparseArray
>>> arr = SparseArray([0, 0, 1, 2])
>>> arr
[0, 0, 1, 2]
Fill: 0
IntIndex
Indices: array([2, 3], dtype=int32)
```

Attributes

| None |

Methods

| None |

<pandas.SparseDtype>(dtype, fill_value) Dtype for data stored in SparseArray.

**pandas.SparseDtype**

class pandas.SparseDtype(dtype=<class 'numpy.float64'>, fill_value=None)

Dtype for data stored in SparseArray.

This dtype implements the pandas ExtensionDtype interface.

**Parameters**

dtype [str, ExtensionDtype, numpy.dtype, type, default numpy.float64] The dtype of the underlying array storing the non-fill value values.

fill_value [scalar, optional] The scalar value not stored in the SparseArray. By default, this depends on dtype.

<table>
<thead>
<tr>
<th>dtype</th>
<th>na_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>np.nan</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>bool</td>
<td>False</td>
</tr>
<tr>
<td>datetime64</td>
<td>pd.NaT</td>
</tr>
<tr>
<td>timedelta64</td>
<td>pd.NaT</td>
</tr>
</tbody>
</table>

The default value may be overridden by specifying a `fill_value`.
The `Series.sparse` accessor may be used to access sparse-specific attributes and methods if the `Series` contains sparse values. See Sparse accessor for more.

### 3.5.10 Text data

When working with text data, where each valid element is a string or missing, we recommend using `StringDtype` (with the alias "string").

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arrays.StringArray(values[, copy])</code></td>
<td>Extension array for string data.</td>
</tr>
<tr>
<td><code>arrays.ArrowStringArray(values)</code></td>
<td>Extension array for string data in a <code>pyarrow</code>.</td>
</tr>
<tr>
<td></td>
<td><code>ChunkedArray</code>.</td>
</tr>
</tbody>
</table>

#### `pandas.arrays.StringArray`

**class** `pandas.arrays.StringArray(values, copy=False)`

Extension array for string data.

New in version 1.0.0.

**Warning:** StringArray is considered experimental. The implementation and parts of the API may change without warning.

**Parameters**

- **values** [array-like] The array of data.
  
  **Warning:** Currently, this expects an object-dtype ndarray where the elements are Python strings or `pandas.NA`. This may change without warning in the future. Use `pandas.array()` with `dtype="string"` for a stable way of creating a `StringArray` from any sequence.

- **copy** [bool, default False] Whether to copy the array of data.

**See also:**

- `array` The recommended function for creating a StringArray.
- `Series.str` The string methods are available on Series backed by a StringArray.
Notes

StringArray returns a BooleanArray for comparison methods.

Examples

```python
>>> pd.array(['This is', 'some text', None, 'data.'], dtype="string")
<StringArray>
['This is', 'some text', <NA>, 'data.]
Length: 4, dtype: string
```

Unlike arrays instantiated with dtype="object", StringArray will convert the values to strings.

```python
>>> pd.array(['1', 1], dtype="object")
<PandasArray>
['1', 1]
Length: 2, dtype: object
>>> pd.array(['1', 1], dtype="string")
<StringArray>
['1', '1']
Length: 2, dtype: string
```

However, instantiating StringArrays directly with non-strings will raise an error.

For comparison methods, StringArray returns a pandas.BooleanArray:

```python
>>> pd.array(['a', None, 'c'], dtype="string") == "a"
<BooleanArray>
[True, <NA>, False]
Length: 3, dtype: boolean
```

Attributes

None

Methods

None

pandas.arrays.ArrowStringArray

class pandas.arrays.ArrowStringArray(values)

Extension array for string data in a pyarrow.ChunkedArray.

New in version 1.2.0.

Warning: ArrowStringArray is considered experimental. The implementation and parts of the API may change without warning.
Parameters

values  [pyarrow.Array or pyarrow.ChunkedArray] The array of data.

See also:
array  The recommended function for creating a ArrowStringArray.
Series.str  The string methods are available on Series backed by a ArrowStringArray.

Notes

ArrowStringArray returns a BooleanArray for comparison methods.

Examples

```python
>>> pd.array(['This is', 'some text', None, 'data.'], dtype="string[pyarrow]")
<ArrowStringArray>
['This is', 'some text', <NA>, 'data.]
Length: 4, dtype: string
```

Attributes

None

Methods

None

StringDtype([storage])  Extension dtype for string data.

pandas.StringDtype

class  pandas.StringDtype(storage=None)

Extension dtype for string data.

New in version 1.0.0.

Warning:  StringDtype is considered experimental. The implementation and parts of the API may change without warning.
In particular, StringDtype.na_value may change to no longer be numpy.nan.

Parameters

storage  ["python", "pyarrow"], optional] If not given, the value of pd.options.mode.
string_storage.
Examples

```python
>>> pd.StringDtype()
string[python]

>>> pd.StringDtype(storage="pyarrow")
string[pyarrow]
```

Attributes

None

Methods

None

The `Series.str` accessor is available for `Series` backed by a `arrays.StringArray`. See `String handling` for more.

### 3.5.11 Boolean data with missing values

The boolean dtype (with the alias "boolean") provides support for storing boolean data (True, False) with missing values, which is not possible with a bool `numpy.ndarray`.

```python
arrays[BooleanArray](values, mask[, copy])
```

Array of boolean (True/False) data with missing values.

**pandas.arrays.BooleanArray**

**class** `pandas.arrays.BooleanArray(values, mask, copy=False)`

Array of boolean (True/False) data with missing values.

This is a pandas Extension array for boolean data, under the hood represented by 2 numpy arrays: a boolean array with the data and a boolean array with the mask (True indicating missing).

BooleanArray implements Kleene logic (sometimes called three-value logic) for logical operations. See `Kleene logical operations` for more.

To construct a BooleanArray from generic array-like input, use `pandas.array()` specifying `dtype="boolean"` (see examples below).

New in version 1.0.0.

**Warning:** BooleanArray is considered experimental. The implementation and parts of the API may change without warning.

**Parameters**

- **values** [numpy.ndarray] A 1-d boolean-dtype array with the data.
**mask**  [numpy.ndarray] A 1-d boolean-dtype array indicating missing values (True indicates missing).

**copy**  [bool, default False] Whether to copy the values and mask arrays.

**Returns**

BooleanArray

**Examples**

Create an BooleanArray with *pandas.array()*:

```python
>>> pd.array([True, False, None], dtype="boolean")
<BooleanArray>
[True, False, <NA>]
Length: 3, dtype: boolean
```

**Attributes**

None

**Methods**

None

---

**BooleanDtype**()  Extension dtype for boolean data.

---

**pandas.BooleanDtype**

**class pandas.BooleanDtype**

Extension dtype for boolean data.

New in version 1.0.0.

**Warning:** BooleanDtype is considered experimental. The implementation and parts of the API may change without warning.

**Examples**

```python
>>> pd.BooleanDtype()
BooleanDtype
```
Attributes

Methods

3.6 Index objects

3.6.1 Index

Many of these methods or variants thereof are available on the objects that contain an index (Series/DataFrame) and those should most likely be used before calling these methods directly.

Index([data, dtype, copy, name, tupleize_cols]) Immutable sequence used for indexing and alignment.

pandas.Index
class pandas.Index(data=None, dtype=None, copy=False, name=None, tupleize_cols=True, **kwargs)

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects.

Parameters

data [array-like (1-dimensional)]
dtype [NumPy dtype (default: object)] If dtype is None, we find the dtype that best fits the data. If an actual dtype is provided, we coerce to that dtype if it’s safe. Otherwise, an error will be raised.
copy [bool] Make a copy of input ndarray.
name [object] Name to be stored in the index.
tupleize_cols [bool (default: True)] When True, attempt to create a MultiIndex if possible.

See also:

RangeIndex Index implementing a monotonic integer range.
CategoricalIndex Index of Categorical s.
MultiIndex A multi-level, or hierarchical Index.
IntervalIndex An Index of Interval s.
DatetimeIndex Index of datetime64 data.
TimedeltaIndex Index of timedelta64 data.
PeriodIndex Index of Period data.
NumericIndex Index of numpy int/uint/float data.
Int64Index Index of purely int64 labels (deprecated).
UInt64Index Index of purely uint64 labels (deprecated).
Float64Index Index of purely float64 labels (deprecated).
Notes
An Index instance can only contain hashable objects

Examples

```python
gt> pd.Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')

gt> pd.Index(list('abc'))
Index(['a', 'b', 'c'], dtype='object')
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Return the transpose, which is by definition self.</td>
</tr>
<tr>
<td><code>array</code></td>
<td>The ExtensionArray of the data backing this Series or Index.</td>
</tr>
<tr>
<td><code>asi8</code></td>
<td>Integer representation of the values.</td>
</tr>
<tr>
<td><code>dtype</code></td>
<td>Return the dtype object of the underlying data.</td>
</tr>
<tr>
<td><code>has_duplicates</code></td>
<td>Check if the Index has duplicate values.</td>
</tr>
<tr>
<td><code>hasnans</code></td>
<td>Return True if there are any NaNs.</td>
</tr>
<tr>
<td><code>inferred_type</code></td>
<td>Return a string of the type inferred from the values.</td>
</tr>
<tr>
<td><code>is_all_dates</code></td>
<td>Whether or not the index values only consist of dates.</td>
</tr>
<tr>
<td><code>is_monotonic</code></td>
<td>Alias for is_monotonic_increasing.</td>
</tr>
<tr>
<td><code>is_monotonic_decreasing</code></td>
<td>Return if the index is monotonic decreasing (only equal or decreasing) values.</td>
</tr>
<tr>
<td><code>is_monotonic_increasing</code></td>
<td>Return if the index is monotonic increasing (only equal or increasing) values.</td>
</tr>
<tr>
<td><code>is_unique</code></td>
<td>Return if the index has unique values.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Return Index or MultiIndex name.</td>
</tr>
<tr>
<td><code>nbytes</code></td>
<td>Return the number of bytes in the underlying data.</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td>Number of dimensions of the underlying data, by definition 1.</td>
</tr>
<tr>
<td><code>nlevels</code></td>
<td>Number of levels.</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>Return a tuple of the shape of the underlying data.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Return the number of elements in the underlying data.</td>
</tr>
<tr>
<td><code>values</code></td>
<td>Return an array representing the data in the Index.</td>
</tr>
</tbody>
</table>

`pandas.Index.T` property

- **Index.T**
  Return the transpose, which is by definition self.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.Index.array

Index.array
The ExtensionArray of the data backing this Series or Index.

Returns
ExtensionArray An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around numpy.ndarray.

.array differs .values which may require converting the data to a different form.

See also:

Index.to_numpy Similar method that always returns a NumPy array.
Series.to_numpy Similar method that always returns a NumPy array.

Notes
This table lays out the different array types for each extension dtype within pandas.

<table>
<thead>
<tr>
<th>dtype</th>
<th>array type</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>Categorical</td>
</tr>
<tr>
<td>period</td>
<td>PeriodArray</td>
</tr>
<tr>
<td>interval</td>
<td>IntervalArray</td>
</tr>
<tr>
<td>IntegerNA</td>
<td>IntegerArray</td>
</tr>
<tr>
<td>string</td>
<td>StringArray</td>
</tr>
<tr>
<td>boolean</td>
<td>BooleanArray</td>
</tr>
<tr>
<td>datetime64[ns, tz]</td>
<td>DatetimeArray</td>
</tr>
</tbody>
</table>

For any 3rd-party extension types, the array type will be an ExtensionArray.
For all remaining dtypes .array will be a arrays.NumpyExtensionArray wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use Series.to_numpy() instead.

Examples
For regular NumPy types like int, and float, a PandasArray is returned.

```python
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned

```python
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
['a', 'b', 'a']
Categories (2, object): ['a', 'b']
```
**pandas.Index.asi8**

**property** `Index.asi8`

Integer representation of the values.

**Returns**

- `ndarray` An ndarray with int64 dtype.

**pandas.Index.dtype**

`Index.dtype`

Return the dtype object of the underlying data.

**pandas.Index.has_duplicates**

**property** `Index.has_duplicates`

Check if the Index has duplicate values.

**Returns**

- `bool` Whether or not the Index has duplicate values.

**Examples**

```python
>>> idx = pd.Index([1, 5, 7, 7])
>>> idx.has_duplicates
True
```

```python
>>> idx = pd.Index([1, 5, 7])
>>> idx.has_duplicates
False
```

```python
>>> idx = pd.Index(["Watermelon", "Orange", "Apple", ...
    "Watermelon"]).astype("category")
>>> idx.has_duplicates
True
```

```python
>>> idx = pd.Index(["Orange", "Apple", ...
    "Watermelon"]).astype("category")
>>> idx.has_duplicates
False
```
pandas.Index.hasnans

Index . hasnans
    Return True if there are any NaNs.
    Enables various performance speedups.

pandas.Index.inferred_type

Index . inferred_type
    Return a string of the type inferred from the values.

pandas.Index.is_all_dates

Index . is_all_dates
    Whether or not the index values only consist of dates.

pandas.Index.is_monotonic

property Index . is_monotonic
    Alias for is_monotonic_increasing.

pandas.Index.is_monotonic_decreasing

property Index . is_monotonic_decreasing
    Return if the index is monotonic decreasing (only equal or decreasing) values.

Examples

```python
>>> Index([3, 2, 1]).is_monotonic_decreasing
True
>>> Index([3, 2, 2]).is_monotonic_decreasing
True
>>> Index([3, 1, 2]).is_monotonic_decreasing
False
```

pandas.Index.is_monotonic_increasing

property Index . is_monotonic_increasing
    Return if the index is monotonic increasing (only equal or increasing) values.
Examples

```python
>>> Index([1, 2, 3]).is_monotonic_increasing
True
>>> Index([1, 2, 2]).is_monotonic_increasing
True
>>> Index([1, 3, 2]).is_monotonic_increasing
False
```

**pandas.Index.is_unique**

`Index.is_unique`

Return if the index has unique values.

**pandas.Index.name**

`property Index.name`

Return Index or MultiIndex name.

**pandas.Index.nbytes**

`property Index.nbytes`

Return the number of bytes in the underlying data.

**pandas.Index.ndim**

`property Index.ndim`

Number of dimensions of the underlying data, by definition 1.

**pandas.Index.nlevels**

`property Index.nlevels`

Number of levels.

**pandas.Index.shape**

`property Index.shape`

Return a tuple of the shape of the underlying data.
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.Index.size**

**property Index.size**

Return the number of elements in the underlying data.

**pandas.Index.values**

**property Index.values**

Return an array representing the data in the Index.

---

**Warning:** We recommend using `Index.array` or `Index.to_numpy()`, depending on whether you need a reference to the underlying data or a NumPy array.

**Returns**

array: numpy.ndarray or ExtensionArray

**See also:**

* `Index.array` Reference to the underlying data.
* `Index.to_numpy` A NumPy array representing the underlying data.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all(*args, **kwags)</td>
<td>Return whether all elements are Truthy.</td>
</tr>
<tr>
<td>any(*args, **kwargs)</td>
<td>Return whether any element is Truthy.</td>
</tr>
<tr>
<td>append(other)</td>
<td>Append a collection of Index options together.</td>
</tr>
<tr>
<td>argmax([axis, skipna])</td>
<td>Return int position of the largest value in the Series.</td>
</tr>
<tr>
<td>argmin([axis, skipna])</td>
<td>Return int position of the smallest value in the Series.</td>
</tr>
<tr>
<td>argsort(*args, **kwargs)</td>
<td>Return the integer indices that would sort the index.</td>
</tr>
<tr>
<td>asof(label)</td>
<td>Return the label from the index, or, if not present, the previous one.</td>
</tr>
<tr>
<td>asof_locs(where, mask)</td>
<td>Return the locations (indices) of labels in the index.</td>
</tr>
<tr>
<td>astype(dtype[, copy])</td>
<td>Create an Index with values cast to dtypes.</td>
</tr>
<tr>
<td>copy([name, deep, dtype, names])</td>
<td>Make a copy of this object.</td>
</tr>
<tr>
<td>delete(loc)</td>
<td>Make new Index with passed location(-s) deleted.</td>
</tr>
<tr>
<td>difference(other[, sort])</td>
<td>Return a new Index with elements of index not in other.</td>
</tr>
<tr>
<td>drop(labels[, errors])</td>
<td>Make new Index with passed list of labels deleted.</td>
</tr>
<tr>
<td>drop_duplicates([keep])</td>
<td>Return Index with duplicate values removed.</td>
</tr>
<tr>
<td>droplevel([level])</td>
<td>Return index with requested level(s) removed.</td>
</tr>
<tr>
<td>dropna([how])</td>
<td>Return Index without NA/NaN values.</td>
</tr>
<tr>
<td>duplicated([keep])</td>
<td>Indicate duplicate index values.</td>
</tr>
<tr>
<td>equals(other)</td>
<td>Determine if two Index object are equal.</td>
</tr>
</tbody>
</table>

continues on next page
Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>factorize()</code></td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><code>fillna(value, downcast)()</code></td>
<td>Fill NA/NaN values with the specified value.</td>
</tr>
<tr>
<td><code>format([name, formatter, na_rep])()</code></td>
<td>Render a string representation of the Index.</td>
</tr>
<tr>
<td><code>get_indexer(target, method, limit, tolerance)</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>get_indexer_for(target)</code></td>
<td>Guaranteed return of an indexer even when non-unique.</td>
</tr>
<tr>
<td><code>get_indexer_non_unique(target)</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>get_level_values(level)</code></td>
<td>Return an Index of values for requested level.</td>
</tr>
<tr>
<td><code>get_loc(key[, method, tolerance])</code></td>
<td>Get integer location, slice or boolean mask for requested label.</td>
</tr>
<tr>
<td><code>get_slice_bound(label, side[, kind])</code></td>
<td>Calculate slice bound that corresponds to given label.</td>
</tr>
<tr>
<td><code>get_value(series, key)</code></td>
<td>Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>groupby(values)</code></td>
<td>Group the index labels by a given array of values.</td>
</tr>
<tr>
<td><code>holds_integer()</code></td>
<td>Whether the type is an integer type.</td>
</tr>
<tr>
<td><code>identical(other)</code></td>
<td>Similar to equals, but checks that object attributes and types are also equal.</td>
</tr>
<tr>
<td><code>insert(loc, item)</code></td>
<td>Make new Index inserting new item at location.</td>
</tr>
<tr>
<td><code>intersection(other[, sort])</code></td>
<td>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td><code>is_(other)</code></td>
<td>More flexible, faster check like <code>is</code> but that works through views.</td>
</tr>
<tr>
<td><code>is_boolean()</code></td>
<td>Check if the Index only consists of booleans.</td>
</tr>
<tr>
<td><code>is_categorical()</code></td>
<td>Check if the Index holds categorical data.</td>
</tr>
<tr>
<td><code>is_floating()</code></td>
<td>Check if the Index is a floating type.</td>
</tr>
<tr>
<td><code>is_integer()</code></td>
<td>Check if the Index only consists of integers.</td>
</tr>
<tr>
<td><code>is_interval()</code></td>
<td>Check if the Index holds Interval objects.</td>
</tr>
<tr>
<td><code>is_mixed()</code></td>
<td>Check if the Index holds data with mixed data types.</td>
</tr>
<tr>
<td><code>is_numeric()</code></td>
<td>Check if the Index only consists of numeric data.</td>
</tr>
<tr>
<td><code>is_object()</code></td>
<td>Check if the Index is of the object dtype.</td>
</tr>
<tr>
<td><code>is_type_compatible(kind)</code></td>
<td>Whether the index type is compatible with the provided type.</td>
</tr>
<tr>
<td><code>isin(values[, level])</code></td>
<td>Return a boolean array where the index values are in <code>values</code>.</td>
</tr>
<tr>
<td><code>isna()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>isnull()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>item()</code></td>
<td>Return the first element of the underlying data as a Python scalar.</td>
</tr>
<tr>
<td><code>join(other[, how, level, return_indexers, sort])</code></td>
<td>Compute join_index and indexers to conform data structures to the new index.</td>
</tr>
<tr>
<td><code>map(mapper[, na_action])</code></td>
<td>Map values using an input mapping or function.</td>
</tr>
<tr>
<td><code>max([axis, skipna])</code></td>
<td>Return the maximum value of the Index.</td>
</tr>
<tr>
<td><code>memory_usage([deep])</code></td>
<td>Memory usage of the values.</td>
</tr>
<tr>
<td><code>min([axis, skipna])</code></td>
<td>Return the minimum value of the Index.</td>
</tr>
<tr>
<td><code>notna()</code></td>
<td>Detect existing (non-missing) values.</td>
</tr>
<tr>
<td><code>notnull()</code></td>
<td>Detect existing (non-missing) values.</td>
</tr>
<tr>
<td><code>nunique([dropna])</code></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><code>putmask(mask, value)</code></td>
<td>Return a new Index of the values set with the mask.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return an ndarray of the flattened values of the underlying data.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reindex</code></td>
<td>Create index with target's values.</td>
</tr>
<tr>
<td><code>rename</code></td>
<td>Alter Index or MultiIndex name.</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>Repeat elements of a Index.</td>
</tr>
<tr>
<td><code>searchsorted</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>set_names</code></td>
<td>Set Index or MultiIndex name.</td>
</tr>
<tr>
<td><code>set_value</code></td>
<td>(DEPRECATED) Fast lookup of value from 1-dimensional ndarray.</td>
</tr>
<tr>
<td><code>shift</code></td>
<td>Shift index by desired number of time frequency increments.</td>
</tr>
<tr>
<td><code>slice_indexer</code></td>
<td>Compute the slice indexer for input labels and step.</td>
</tr>
<tr>
<td><code>slice_locs</code></td>
<td>Compute slice locations for input labels.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Use sort_values instead.</td>
</tr>
<tr>
<td><code>sort_values</code></td>
<td>Return a sorted copy of the index.</td>
</tr>
<tr>
<td><code>sortlevel</code></td>
<td>For internal compatibility with the Index API.</td>
</tr>
<tr>
<td><code>str</code></td>
<td>alias of pandas.core.strings.accessor.StringMethods</td>
</tr>
<tr>
<td><code>symmetric_difference</code></td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Return a new Index of the values selected by the indices.</td>
</tr>
<tr>
<td><code>to_flat_index</code></td>
<td>Identity method.</td>
</tr>
<tr>
<td><code>to_frame</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td><code>to_list</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>to_native_types</code></td>
<td>(DEPRECATED) Format specified values of self and return them.</td>
</tr>
<tr>
<td><code>to_numpy</code></td>
<td>A NumPy ndarray representing the values in this Series or Index.</td>
</tr>
<tr>
<td><code>to_series</code></td>
<td>Create a Series with both index and values equal to the index keys.</td>
</tr>
<tr>
<td><code>tolist</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>transpose</code></td>
<td>Return the transpose, which is by definition self.</td>
</tr>
<tr>
<td><code>union</code></td>
<td>Form the union of two Index objects.</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>Return unique values in the index.</td>
</tr>
<tr>
<td><code>value_counts</code></td>
<td>Return a Series containing counts of unique values.</td>
</tr>
<tr>
<td><code>where</code></td>
<td>Replace values where the condition is False.</td>
</tr>
</tbody>
</table>

#### pandas.Index.all

`Index.all(*args, **kwags)`

Return whether all elements are Truthy.

**Parameters**

- *args Required for compatibility with numpy.
- **kwargs Required for compatibility with numpy.

**Returns**

- all [bool or array-like (if axis is specified)] A single element array-like may be converted to bool.
See also:

- **Index.any** Return whether any element in an Index is True.
- **Series.any** Return whether any element in a Series is True.
- **Series.all** Return whether all elements in a Series are True.

**Notes**

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

**Examples**

True, because nonzero integers are considered True.

```
>>> pd.Index([1, 2, 3]).all()
True
```

False, because 0 is considered False.

```
>>> pd.Index([0, 1, 2]).all()
False
```

**pandas.Index.any**

```python
Index.any(*args, **kwargs)
```

Return whether any element is Truthy.

**Parameters**

- `*args` Required for compatibility with numpy.
- `**kwargs` Required for compatibility with numpy.

**Returns**

- `any` [bool or array-like (if axis is specified)] A single element array-like may be converted to bool.

See also:

- **Index.all** Return whether all elements are True.
- **Series.all** Return whether all elements are True.
Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```python
>>> index = pd.Index([0, 1, 2])
>>> index.any()
True

>>> index = pd.Index([0, 0, 0])
>>> index.any()
False
```

**pandas.Index.append**

Index.append(other)

Append a collection of Index options together.

- **Parameters**
  - other [Index or list/tuple of indices]

- **Returns**
  - Index

**pandas.Index.argmax**

Index.argmax(axis=None, skipna=True, *args, **kwargs)

Return int position of the largest value in the Series.

If the maximum is achieved in multiple locations, the first row position is returned.

- **Parameters**
  - axis {None} Dummy argument for consistency with Series.
  - skipna [bool, default True] Exclude NA/null values when showing the result.
  - *args, **kwargs Additional arguments and keywords for compatibility with NumPy.

- **Returns**
  - int Row position of the maximum value.

See also:

- **Series.argmax** Return position of the maximum value.
- **Series.argmin** Return position of the minimum value.
- **numpy.ndarray.argmax** Equivalent method for numpy arrays.
- **Series.idxmax** Return index label of the maximum values.
- **Series.idxmin** Return index label of the minimum values.
Examples

Consider dataset containing cereal calories

```python
>>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
                  'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
```

```python
>>> s
Corn Flakes         100.0
Almond Delight      110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff          110.0
dtype: float64
```

```python
>>> s.argmax()
2
>>> s.argmin()
0
```

The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

`pandas.Index.argmin`

`Index.argmin`(axis=None, skipna=True, *args, **kwargs)

Return int position of the smallest value in the Series.

If the minimum is achieved in multiple locations, the first row position is returned.

**Parameters**

- **axis** ([None]) Dummy argument for consistency with Series.
- **skipna** [bool, default True] Exclude NA/null values when showing the result.
- ***args, **kwargs** Additional arguments and keywords for compatibility with NumPy.

**Returns**

- **int** Row position of the minimum value.

**See also:**

- `Series.argmin` Return position of the minimum value.
- `Series.argmax` Return position of the maximum value.
- `numpy.ndarray.argmin` Equivalent method for numpy arrays.
- `Series.idxmax` Return index label of the maximum values.
- `Series.idxmin` Return index label of the minimum values.
**Examples**

Consider dataset containing cereal calories

```python
>>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
...                'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes    100.0
Almond Delight 110.0
Cinnamon Toast Crunch    120.0
Cocoa Puff    110.0
dtype: float64
```

```python
>>> s.argmax()
2
>>> s.argmin()
0
```

The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

**pandas.Index.argsort**

`Index.argsort(*args, **kwargs)`

Return the integer indices that would sort the index.

**Parameters**

* `*args` Passed to `numpy.ndarray.argsort`.
* `**kwargs` Passed to `numpy.ndarray.argsort`.

**Returns**

`np.ndarray[np.intp]` Integer indices that would sort the index if used as an indexer.

**See also:**

`numpy.argsort` Similar method for NumPy arrays.

`Index.sort_values` Return sorted copy of Index.

**Examples**

```python
>>> idx = pd.Index(['b', 'a', 'd', 'c'])
>>> idx
Index(['b', 'a', 'd', 'c'], dtype='object')
```

```python
>>> order = idx.argsort()
>>> order
array([1, 0, 3, 2])
```

```python
>>> idx[order]
Index(['a', 'b', 'c', 'd'], dtype='object')
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.Index.asof
final Index.asof(label)
Return the label from the index, or, if not present, the previous one.
Assuming that the index is sorted, return the passed index label if it is in the index, or return the previous
index label if the passed one is not in the index.
Parameters
label [object] The label up to which the method returns the latest index label.
Returns
object The passed label if it is in the index. The previous label if the passed label is not
in the sorted index or NaN if there is no such label.
See also:
Series.asof Return the latest value in a Series up to the passed index.
merge_asof Perform an asof merge (similar to left join but it matches on nearest key rather than equal
key).
Index.get_loc An asof is a thin wrapper around get_loc with method=’pad’.
Examples
Index.asof returns the latest index label up to the passed label.
>>> idx = pd.Index(['2013-12-31', '2014-01-02', '2014-01-03'])
>>> idx.asof('2014-01-01')
'2013-12-31'
If the label is in the index, the method returns the passed label.
>>> idx.asof('2014-01-02')
'2014-01-02'
If all of the labels in the index are later than the passed label, NaN is returned.
>>> idx.asof('1999-01-02')
nan
If the index is not sorted, an error is raised.
>>> idx_not_sorted = pd.Index(['2013-12-31', '2015-01-02',
...
'2014-01-03'])
>>> idx_not_sorted.asof('2013-12-31')
Traceback (most recent call last):
ValueError: index must be monotonic increasing or decreasing

3.6. Index objects

2137


**pandas.Index.asof_locs**

Index.asof_locs(where, mask)

Return the locations (indices) of labels in the index.

As in the `asof` function, if the label (a particular entry in `where`) is not in the index, the latest index label up to the passed label is chosen and its index returned.

If all of the labels in the index are later than a label in `where`, -1 is returned.

`mask` is used to ignore NA values in the index during calculation.

**Parameters**

- **where** [Index] An Index consisting of an array of timestamps.
- **mask** [np.ndarray[bool]] Array of booleans denoting where values in the original data are not NA.

**Returns**

np.ndarray[np.intp] An array of locations (indices) of the labels from the Index which correspond to the return values of the `asof` function for every element in `where`.

**pandas.Index.astype**

Index.astype(dtype, copy=True)

Create an Index with values cast to dtypes.

The class of a new Index is determined by `dtype`. When conversion is impossible, a TypeError exception is raised.

**Parameters**

- **dtype** [numpy dtype or pandas type] Note that any signed integer `dtype` is treated as 'int64', and any unsigned integer `dtype` is treated as 'uint64', regardless of the size.
- **copy** [bool, default True] By default, `astype` always returns a newly allocated object. If `copy` is set to False and internal requirements on `dtype` are satisfied, the original data is used to create a new Index or the original Index is returned.

**Returns**

Index Index with values cast to specified `dtype`.

**pandas.Index.copy**

Index.copy(name=None, deep=False, dtype=None, names=None)

Make a copy of this object.

Name and `dtype` sets those attributes on the new object.

**Parameters**

- **name** [Label, optional] Set name for new object.
- **deep** [bool, default False]
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

**dtype** [numpy dtype or pandas type, optional] Set dtype for new object.  
Deprecated since version 1.2.0: use `astype` method instead.

**names** [list-like, optional] Kept for compatibility with MultiIndex. Should not be used.  
Deprecated since version 1.4.0: use `name` instead.

**Returns**

- **Index** Index refer to new object which is a copy of this object.

**Notes**

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy.

**pandas.Index.delete**

Index.delete(loc)

Make new Index with passed location(-s) deleted.

**Parameters**

- **loc** [int or list of int] Location of item(-s) which will be deleted. Use a list of locations to delete more than one value at the same time.

**Returns**

- **Index** Will be same type as self, except for RangeIndex.

**See also:**

**numpy.delete** Delete any rows and column from NumPy array (ndarray).

**Examples**

```python
>>> idx = pd.Index(['a', 'b', 'c'])
>>> idx.delete(1)
Index(['a', 'c'], dtype='object')

>>> idx = pd.Index(['a', 'b', 'c'])
>>> idx.delete([0, 2])
Index(['b'], dtype='object')
```

**pandas.Index.difference**

**final** Index.difference(\textit{other}, \textit{sort}=\texttt{None})

Return a new Index with elements of index not in \textit{other}.

This is the set difference of two Index objects.

**Parameters**

- **other** [Index or array-like]
sort [False or None, default None] Whether to sort the resulting index. By default, the values are attempted to be sorted, but any TypeError from incomparable elements is caught by pandas.

- None: Attempt to sort the result, but catch any TypeErrors from comparing incomparable elements.
- False: Do not sort the result.

Returns
difference [Index]

Examples

```python
>>> idx1 = pd.Index([2, 1, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
>>> idx1.difference(idx2, sort=False)
Int64Index([2, 1], dtype='int64')
```

**pandas.Index.drop**

`Index.drop(labels, errors='raise')`

Make new Index with passed list of labels deleted.

Parameters

- **labels** [array-like or scalar]
- **errors** [['ignore', 'raise'], default 'raise'] If 'ignore', suppress error and existing labels are dropped.

Returns
dropped [Index] Will be same type as self, except for RangeIndex.

Raises

- **KeyError** If not all of the labels are found in the selected axis

**pandas.Index.drop_duplicates**

`Index.drop_duplicates(keep='first')`

Return Index with duplicate values removed.

Parameters

- **keep** [['first', 'last', False], default 'first']
  - 'first': Drop duplicates except for the first occurrence.
  - 'last': Drop duplicates except for the last occurrence.
  - False: Drop all duplicates.

Returns
deduplicated [Index]

See also:

Series.drop_duplicates Equivalent method on Series.
DataFrame.drop_duplicates Equivalent method on DataFrame.
Index.duplicated Related method on Index, indicating duplicate Index values.

Examples

Generate an pandas.Index with duplicate values.

```python
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'])
```

The `keep` parameter controls which duplicate values are removed. The value ‘first’ keeps the first occurrence for each set of duplicated entries. The default value of keep is ‘first’.

```python
>>> idx.drop_duplicates(keep='first')
Index(['lama', 'cow', 'beetle', 'hippo'], dtype='object')
```

The value ‘last’ keeps the last occurrence for each set of duplicated entries.

```python
>>> idx.drop_duplicates(keep='last')
Index(['cow', 'beetle', 'lama', 'hippo'], dtype='object')
```

The value False discards all sets of duplicated entries.

```python
>>> idx.drop_duplicates(keep=False)
Index(['cow', 'beetle', 'hippo'], dtype='object')
```

pandas.Index.droplevel

final Index.droplevel(level=0)

Return index with requested level(s) removed.

If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

Parameters

level [int, str, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

Returns

Index or MultiIndex
Examples

```python
>>> mi = pd.MultiIndex.from_arrays(
... [[1, 2], [3, 4], [5, 6]], names=['x', 'y', 'z'])
```
```
>>> mi
MultiIndex([(1, 3, 5),
            (2, 4, 6)],
           names=['x', 'y', 'z'])
```
```
>>> mi.droplevel()
MultiIndex([(3, 5),
            (4, 6)],
           names=['y', 'z'])
```
```
>>> mi.droplevel(2)
MultiIndex([(1, 3),
            (2, 4)],
           names=['x', 'y'])
```
```
>>> mi.droplevel('z')
MultiIndex([(1, 3),
            (2, 4)],
           names=['x', 'y'])
```
```
>>> mi.droplevel(['x', 'y'])
Int64Index([5, 6], dtype='int64', name='z')
```

**pandas.Index.dropna**

Index.dropna(**how='any'**)

Return Index without NA/NaN values.

**Parameters**

- **how** [{'any', 'all'}, default 'any'] If the Index is a MultiIndex, drop the value when any or all levels are NaN.

**Returns**

Index

**pandas.Index.duplicated**

Index.duplicated(**keep='first'**)

Indicate duplicate index values.

Duplicated values are indicated as True values in the resulting array. Either all duplicates, all except the first, or all except the last occurrence of duplicates can be indicated.

**Parameters**

- **keep** [{'first', 'last', False}, default 'first'] The value or values in a set of duplicates to mark as missing.
• ‘first’ : Mark duplicates as True except for the first occurrence.
• ‘last’ : Mark duplicates as True except for the last occurrence.
• False : Mark all duplicates as True.

Returns

np.ndarray[bool]

See also:

Series.duplicated Equivalent method on pandas.Series.
DataFrame.duplicated Equivalent method on pandas.DataFrame.
Index.drop_duplicates Remove duplicate values from Index.

Examples

By default, for each set of duplicated values, the first occurrence is set to False and all others to True:

```python
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> idx.duplicated()
array([False, False, True, False, True])
```

which is equivalent to

```python
>>> idx.duplicated(keep='first')
array([False, False, True, False, True])
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```python
>>> idx.duplicated(keep='last')
array([ True, False, True, False, False])
```

By setting keep on False, all duplicates are True:

```python
>>> idx.duplicated(keep=False)
array([ True, False, True, False, True])
```

pandas.Index.equals

Index.equals(other)

Determine if two Index object are equal.

The things that are being compared are:

• The elements inside the Index object.
• The order of the elements inside the Index object.

Parameters

other [Any] The other object to compare against.

Returns
**bool**  True if “other” is an Index and it has the same elements and order as the calling index; False otherwise.

**Examples**

```python
>>> idx1 = pd.Index([1, 2, 3])
>>> idx1
Int64Index([1, 2, 3], dtype='int64')
>>> idx1.equals(pd.Index([1, 2, 3]))
True
```

The elements inside are compared

```python
>>> idx2 = pd.Index(["1", "2", "3"])  
>>> idx2
Index(["1", "2", "3"], dtype='object')
```

```python
>>> idx1.equals(idx2)
False
```

The order is compared

```python
>>> ascending_idx = pd.Index([1, 2, 3])
>>> ascending_idx
Int64Index([1, 2, 3], dtype='int64')
```

```python
>>> descending_idx = pd.Index([3, 2, 1])
>>> descending_idx
Int64Index([3, 2, 1], dtype='int64')
>>> ascending_idx.equals(descending_idx)
False
```

The dtype is not compared

```python
>>> int64_idx = pd.Int64Index([1, 2, 3])
>>> int64_idx
Int64Index([1, 2, 3], dtype='int64')
```

```python
>>> uint64_idx = pd.UInt64Index([1, 2, 3])
>>> uint64_idx
UInt64Index([1, 2, 3], dtype='uint64')
>>> int64_idx.equals(uint64_idx)
True
```

**pandas.Index.factorize**

```python
Index.factorize(sort=False, na_sentinel=-1)
```

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. `factorize` is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

**Parameters**
sort [bool, default False] Sort uniques and shuffle codes to maintain the relationship.

na_sentinel [int or None, default -1] Value to mark “not found”. If None, will not drop
the NaN from the uniques of the values.

Changed in version 1.1.2.

Returns

codes [ndarray] An integer ndarray that’s an indexer into uniques. uniques.
take(codes) will have the same values as values.

uniques [ndarray, Index, or Categorical] The unique valid values. When values is Cat-
egorical, uniques is a Categorical. When values is some other pandas object, an
Index is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there’s a missing value in values, uniques will not contain an entry
for it.

See also:

cut Discretize continuous-valued array.
unique Find the unique value in an array.

Notes

Reference the user guide for more examples.

Examples

These examples all show factorize as a top-level method like pd.factorize(values). The results are
identical for methods like Series.factorize().

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> codes
array([0, 0, 1, 2, 0]...)
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With sort=True, the uniques will be sorted, and codes will be shuffled so that the relationship is the
maintained.

```python
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> codes
array([1, 1, 0, 2, 1]...)
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in codes with na_sentinel (-1 by default). Note that missing values are never
included in uniques.
Thus far, we’ve only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of `uniques` will differ. For Categoricals, a `Categorical` is returned.

```python
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1]...)
>>> uniques
['a', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Notice that 'b' is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```python
>>> cat = pd.Series(['a', 'a', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1]...)
>>> uniques
Index(['a', 'c'], dtype='object')
```

If NaN is in the values, and we want to include NaN in the uniques of the values, it can be achieved by setting `na_sentinel=None`.

```python
>>> values = np.array([1, 2, 1, np.nan])
>>> codes, uniques = pd.factorize(values)  # default: na_sentinel=-1
>>> codes
array([0, 1, 0, -1])
>>> uniques
array([1., 2.])
```

```python
>>> codes, uniques = pd.factorize(values, na_sentinel=None)
>>> codes
array([0, 1, 0, 2])
>>> uniques
array([1., 2., nan])
```
pandas.Index.fillna

Index.fillna(value=None, downcast=None)
Fill NA/NaN values with the specified value.

Parameters

value  [scalar] Scalar value to use to fill holes (e.g. 0). This value cannot be a list-likes.

downcast  [dict, default is None] A dict of item->dtype of what to downcast if possible, or the string ‘infer’ which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Index

See also:

**DataFrame.fillna** Fill NaN values of a DataFrame.

**Series.fillna** Fill NaN Values of a Series.

pandas.Index.format

Index.format(name=False, formatter=None, na_rep='NaN')
Render a string representation of the Index.

pandas.Index.get_indexer

final Index.get_indexer(target, method=None, limit=None, tolerance=None)
Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target  [Index]

method  [[None, ‘pad’/’ffill’, ‘backfill’/’bfill’, ‘nearest’], optional]
  • default: exact matches only.
  • pad / ffill: find the PREVIOUS index value if no exact match.
  • backfill / bfill: use NEXT index value if no exact match
  • nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit  [int, optional] Maximum number of consecutive labels in target to match for inexact matches.

tolerance  [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation abs(index[indexer] - target) <= tolerance.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.
Returns

indexer [np.ndarray[np.intp]] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Notes

Returns -1 for unmatched values, for further explanation see the example below.

Examples

```python
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1, 2, -1])
```

Notice that the return value is an array of locations in index and x is marked by -1, as it is not in index.

pandas.Index.get_indexer_for

```python
final Index.get_indexer_for(target)
```

Guaranteed return of an indexer even when non-unique.

This dispatches to get_indexer or get_indexer_non_unique as appropriate.

Returns

np.ndarray[np.intp] List of indices.

Examples

```python
>>> idx = pd.Index([np.nan, 'var1', np.nan])
>>> idx.get_indexer_for([np.nan])
array([ 0, 2])
```

pandas.Index.get_indexer_non_unique

```python
Index.get_indexer_non_unique(target)
```

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [Index]

Returns

indexer [np.ndarray[np.intp]] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

missing [np.ndarray[np.intp]] An indexer into the target of the values not found. These correspond to the -1 in the indexer array.
pandas.Index.get_level_values

Index.get_level_values(level)

Return an Index of values for requested level.

This is primarily useful to get an individual level of values from a MultiIndex, but is provided on Index as well for compatibility.

Parameters

level [int or str] It is either the integer position or the name of the level.

Returns

Index Calling object, as there is only one level in the Index.

See also:

MultiIndex.get_level_values Get values for a level of a MultiIndex.

Notes

For Index, level should be 0, since there are no multiple levels.

Examples

```python
>>> idx = pd.Index(list('abc'))
>>> idx
Index(['a', 'b', 'c'], dtype='object')

Get level values by supplying level as integer:

```python
>>> idx.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object')
```

pandas.Index.get_loc

Index.get_loc(key, method=None, tolerance=None)

Get integer location, slice or boolean mask for requested label.

Parameters

key [label]

method [[None, ‘pad’/’ffill’, ‘backfill’/’bfill’, ‘nearest’], optional]

• default: exact matches only.
• pad / ffill: find the PREVIOUS index value if no exact match.
• backfill / bfill: use NEXT index value if no exact match
• nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.
tolerance [int or float, optional] Maximum distance from index value for inexact matches. The value of the index at the matching location must satisfy the equation \( \text{abs(index[loc] - key)} \leq \text{tolerance} \).

**Returns**

loc [int if unique index, slice if monotonic index, else mask]

**Examples**

```python
>>> unique_index = pd.Index(list('abc'))
>>> unique_index.get_loc('b')
1

>>> monotonic_index = pd.Index(list('abbc'))
>>> monotonic_index.get_loc('b')
slice(1, 3, None)

>>> non_monotonic_index = pd.Index(list('abcb'))
>>> non_monotonic_index.get_loc('b')
array([False, True, False, True])
```

**pandas.Index.get_slice_bound**

Index.get_slice_bound(label, side, kind=NoDefault.no_default)

Calculate slice bound that corresponds to given label.

**Parameters**

- label [object]
- side [‘left’, ‘right’]
- kind [‘loc’, ‘getitem’ or None] Deprecated since version 1.4.0.

**Returns**

- int Index of label.

**pandas.Index.get_value**

Index.get_value(series, key)

Fast lookup of value from 1-dimensional ndarray.

Only use this if you know what you’re doing.

**Returns**

- scalar or Series
**pandas.Index.groupby**

```python
final Index.groupby(values)
```
Group the index labels by a given array of values.

**Parameters**
- **values** [array] Values used to determine the groups.

**Returns**
- **dict** {group name -> group labels}

**pandas.Index.holds_integer**

```python
final Index.holds_integer()
```
Whether the type is an integer type.

**pandas.Index.identical**

```python
final Index.identical(other)
```
Similar to equals, but checks that object attributes and types are also equal.

**Returns**
- **bool** If two Index objects have equal elements and same type True, otherwise False.

**pandas.Index.insert**

```python
Index.insert(loc, item)
```
Make new Index inserting new item at location.

Follows Python numpy.insert semantics for negative values.

**Parameters**
- **loc** [int]
- **item** [object]

**Returns**
- **new_index** [Index]

**pandas.Index.intersection**

```python
final Index.intersection(other, sort=False)
```
Form the intersection of two Index objects.

This returns a new Index with elements common to the index and `other`.

**Parameters**
- **other** [Index or array-like]
- **sort** [False or None, default False] Whether to sort the resulting index.
• False: do not sort the result.
• None: sort the result, except when self and other are equal or when the values cannot be compared.

**Returns**

intersection [Index]

**Examples**

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.intersection(idx2)
Int64Index([3, 4], dtype='int64')
```

**pandas.Index.is_**

**final** Index.is_(other)

More flexible, faster check like is but that works through views.

Note: this is not the same as `Index.identical()`, which checks that metadata is also the same.

**Parameters**

other [object] Other object to compare against.

**Returns**

bool True if both have same underlying data, False otherwise.

**See also:**

`Index.identical` Works like `Index.is_` but also checks metadata.

**pandas.Index.is_boolean**

**final** Index.is_boolean()

Check if the Index only consists of booleans.

**Returns**

bool Whether or not the Index only consists of booleans.

**See also:**

`is_integer` Check if the Index only consists of integers.
`is_floating` Check if the Index is a floating type.
`is_numeric` Check if the Index only consists of numeric data.
`is_object` Check if the Index is of the object dtype.
`is_categorical` Check if the Index holds categorical data.
`is_interval` Check if the Index holds Interval objects.
`is_mixed` Check if the Index holds data with mixed data types.
Examples

```python
>>> idx = pd.Index([True, False, True])
>>> idx.is_boolean()
True

>>> idx = pd.Index(["True", "False", "True"])
>>> idx.is_boolean()
False

>>> idx = pd.Index([True, False, "True"])
>>> idx.is_boolean()
False
```

**pandas.Index.is_categorical**

**final** `Index.is_categorical()`

Check if the Index holds categorical data.

**Returns**

`bool` True if the Index is categorical.

**See also:**

- `CategoricalIndex` Index for categorical data.
- `is_boolean` Check if the Index only consists of booleans.
- `is_integer` Check if the Index only consists of integers.
- `is_floating` Check if the Index is a floating type.
- `is_numeric` Check if the Index only consists of numeric data.
- `is_object` Check if the Index is of the object dtype.
- `is_interval` Check if the Index holds Interval objects.
- `is_mixed` Check if the Index holds data with mixed data types.

Examples

```python
>>> idx = pd.Index(["Watermelon", "Orange", "Apple", ... "Watermelon"]).astype("category")
>>> idx.is_categorical()
True

>>> idx = pd.Index([1, 3, 5, 7])
>>> idx.is_categorical()
False
```
```python
>>> s = pd.Series(['Peter', 'Victor', 'Elisabeth', 'Mar'])
>>> s
0   Peter
1   Victor
2  Elisabeth
3    Mar
dtype: object
>>> s.index.is_categorical()
False
```

**pandas.Index.is_floating**

**final Index.is_floating()**

Check if the Index is a floating type.

The Index may consist of only floats, NaNs, or a mix of floats, integers, or NaNs.

**Returns**

*bool* Whether or not the Index only consists of only consists of floats, NaNs, or a mix of floats, integers, or NaNs.

**See also:**

- **is_boolean** Check if the Index only consists of booleans.
- **is_integer** Check if the Index only consists of integers.
- **is_numeric** Check if the Index only consists of numeric data.
- **is_object** Check if the Index is of the object dtype.
- **is_categorical** Check if the Index holds categorical data.
- **is_interval** Check if the Index holds Interval objects.
- **is_mixed** Check if the Index holds data with mixed data types.

**Examples**

```python
>>> idx = pd.Index([1.0, 2.0, 3.0, 4.0])
>>> idx.is_floating()
True

>>> idx = pd.Index([1.0, 2.0, np.nan, 4.0])
>>> idx.is_floating()
True

>>> idx = pd.Index([1, 2, 3, 4, np.nan])
>>> idx.is_floating()
True

>>> idx = pd.Index([1, 2, 3, 4])
>>> idx.is_floating()
False
```
**pandas.Index.is_integer**

**final Index.is_integer()**

Check if the Index only consists of integers.

**Returns**

`bool` Whether or not the Index only consists of integers.

**See also:**

- `is_boolean` Check if the Index only consists of booleans.
- `is_floating` Check if the Index is a floating type.
- `is_numeric` Check if the Index only consists of numeric data.
- `is_object` Check if the Index is of the object dtype.
- `is_categorical` Check if the Index holds categorical data.
- `is_interval` Check if the Index holds Interval objects.
- `is_mixed` Check if the Index holds data with mixed data types.

**Examples**

```python
>>> idx = pd.Index([1, 2, 3, 4])
>>> idx.is_integer()
True

>>> idx = pd.Index([1.0, 2.0, 3.0, 4.0])
>>> idx.is_integer()
False

>>> idx = pd.Index(["Apple", "Mango", "Watermelon"])  
>>> idx.is_integer()
False
```

**pandas.Index.is_interval**

**final Index.is_interval()**

Check if the Index holds Interval objects.

**Returns**

`bool` Whether or not the Index holds Interval objects.

**See also:**

- `IntervalIndex` Index for Interval objects.
- `is_boolean` Check if the Index only consists of booleans.
- `is_integer` Check if the Index only consists of integers.
- `is_floating` Check if the Index is a floating type.
is_numeric  Check if the Index only consists of numeric data.

is_object  Check if the Index is of the object dtype.

is_categorical  Check if the Index holds categorical data.

is_mixed  Check if the Index holds data with mixed data types.

Examples

```python
>>> idx = pd.Index([pd.Interval(left=0, right=5),
                  ...                pd.Interval(left=5, right=10)])
>>> idx.is_interval()
True
```

```python
>>> idx = pd.Index([1, 3, 5, 7])
>>> idx.is_interval()
False
```

pandas.Index.is_mixed

final  Index.is_mixed()
Check if the Index holds data with mixed data types.

Returns

bool  Whether or not the Index holds data with mixed data types.

See also:

is_boolean  Check if the Index only consists of booleans.

is_integer  Check if the Index only consists of integers.

is_floating  Check if the Index is a floating type.

is_numeric  Check if the Index only consists of numeric data.

is_object  Check if the Index is of the object dtype.

is_categorical  Check if the Index holds categorical data.

is_interval  Check if the Index holds Interval objects.

Examples

```python
>>> idx = pd.Index(['a', np.nan, 'b'])
>>> idx.is_mixed()
True
```

```python
>>> idx = pd.Index([1.0, 2.0, 3.0, 5.0])
>>> idx.is_mixed()
False
```
pandas.Index.is_numeric

**final Index.is_numeric()**

Check if the Index only consists of numeric data.

**Returns**

```
bool Whether or not the Index only consists of numeric data.
```

**See also:**

*is_boolean* Check if the Index only consists of booleans.

*is_integer* Check if the Index only consists of integers.

*is_floating* Check if the Index is a floating type.

*is_object* Check if the Index is of the object dtype.

*is_categorical* Check if the Index holds categorical data.

*is_interval* Check if the Index holds Interval objects.

*is_mixed* Check if the Index holds data with mixed data types.

**Examples**

```python
>>> idx = pd.Index([1.0, 2.0, 3.0, 4.0])
>>> idx.is_numeric()
True

>>> idx = pd.Index([1, 2, 3, 4.0])
>>> idx.is_numeric()
True

>>> idx = pd.Index([1, 2, 3, 4])
>>> idx.is_numeric()
True

>>> idx = pd.Index([1, 2, 3, 4.0, np.nan])
>>> idx.is_numeric()
True

>>> idx = pd.Index([1, 2, 3, 4.0, np.nan, "Apple"])
>>> idx.is_numeric()
False
```
pandas.Index.is_object

**final Index.is_object()**
Check if the Index is of the object dtype.

**Returns**

`bool` Whether or not the Index is of the object dtype.

**See also:**

- `is_boolean` Check if the Index only consists of booleans.
- `is_integer` Check if the Index only consists of integers.
- `is_float` Check if the Index is a floating type.
- `is_numeric` Check if the Index only consists of numeric data.
- `is_categorical` Check if the Index holds categorical data.
- `is_interval` Check if the Index holds Interval objects.
- `is_mixed` Check if the Index holds data with mixed data types.

**Examples**

```python
>>> idx = pd.Index(['Apple', 'Mango', 'Watermelon'])
>>> idx.is_object()
True

>>> idx = pd.Index(['Apple', 'Mango', 2.0])
>>> idx.is_object()
True

>>> idx = pd.Index(['Watermelon', 'Orange', 'Apple', ...
                          'Watermelon']).astype('category')
>>> idx.is_object()
False

>>> idx = pd.Index([1.0, 2.0, 3.0, 4.0])
>>> idx.is_object()
False
```

pandas.Index.is_type_compatible

**Index.is_type_compatible(kind)**
Whether the index type is compatible with the provided type.
pandas.Index.isin

Index.isin(values, level=None)
Return a boolean array where the index values are in values.
Compute boolean array of whether each index value is found in the passed set of values. The length of the
returned boolean array matches the length of the index.

Parameters
values [set or list-like] Sought values.
level [str or int, optional] Name or position of the index level to use (if the index is a
MultiIndex).

Returns
np.ndarray[bool] NumPy array of boolean values.

See also:
Series.isin Same for Series.
DataFrame.isin Same method for DataFrames.

Notes
In the case of MultiIndex you must either specify values as a list-like object containing tuples that are the
same length as the number of levels, or specify level. Otherwise it will raise a ValueError.
If level is specified:
• if it is the name of one and only one index level, use that level;
• otherwise it should be a number indicating level position.

Examples

```python
>>> idx = pd.Index([1, 2, 3])
>>> idx
Int64Index([1, 2, 3], dtype='int64')
```
Check whether each index value in a list of values.

```python
>>> idx.isin([1, 4])
array([True, False, False])
```

```python
>>> midx = pd.MultiIndex.from_arrays([[1, 2, 3],
... ['red', 'blue', 'green']],
... names=('number', 'color'))
```
Check whether the strings in the ‘color’ level of the MultiIndex are in a list of colors.
To check across the levels of a MultiIndex, pass a list of tuples:

```python
>>> midx.isin([(1, 'red'), (3, 'red')])
array([ True, False, False])
```

For a DatetimeIndex, string values in `values` are converted to Timestamps.

```python
>>> dates = ['2000-03-11', '2000-03-12', '2000-03-13']
>>> dti = pd.to_datetime(dates)
>>> dti
DatetimeIndex(['2000-03-11', '2000-03-12', '2000-03-13'],
              dtype='datetime64[ns]', freq=None)
>>> dti.isin(['2000-03-11'])
array([ True, False, False])
```

### pandas.Index.isna

**final** `Index.isna()`  
Detect missing values.  

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns**

`numpy.ndarray[bool]` A boolean array of whether my values are NA.

**See also:**

- `Index.notna` Boolean inverse of isna.
- `Index.dropna` Omit entries with missing values.
- `isna` Top-level isna.
- `Series.isna` Detect missing values in Series object.

**Examples**

Show which entries in a pandas.Index are NA. The result is an array.

```python
>>> idx = pd.Index([5.2, 6.0, np.nan])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False, True])
```

Empty strings are not considered NA values. None is considered an NA value.
For datetimes, NaT (Not a Time) is considered as an NA value.

```python
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
                          pd.Timestamp(''),
                          None,
                          pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
               dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False, True, True, True])
```

### pandas.Index.isnull

**Index.isnull()**

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings `'` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns**

`numpy.ndarray[bool]` A boolean array of whether my values are NA.

**See also:**

- `Index.notna` Boolean inverse of isna.
- `Index.dropna` Omit entries with missing values.
- `isna` Top-level isna.
- `Series.isna` Detect missing values in Series object.

**Examples**

Show which entries in a pandas.Index are NA. The result is an array.

```python
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False, True])
```

Empty strings are not considered NA values. None is considered an NA value.
```python
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False, True])
```

For datetimes, `NaT` (Not a Time) is considered as an NA value.

```python
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'), ...
...                         pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
               dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False, True, True, True])
```

### pandas.Index.item

**Index.item()**

Return the first element of the underlying data as a Python scalar.

**Returns**

- **scalar**: The first element of `%klass`.

**Raises**

- **ValueError**: If the data is not length-1.

### pandas.Index.join

**final Index.join(other, how='left', level=None, return_indexers=False, sort=False)**

Compute join_index and indexers to conform data structures to the new index.

**Parameters**

- **other**: [Index]
- **how**: ['left', 'right', 'inner', 'outer']
- **level**: [int or level name, default None]
- **return_indexers**: [bool, default False]
- **sort**: [bool, default False] Sort the join keys lexicographically in the result Index. If False, the order of the join keys depends on the join type (how keyword).

**Returns**

- **join_index**, (left_indexer, right_indexer)
pandas.Index.map

Index.map(mapper, na_action=None)
Map values using an input mapping or function.

Parameters

mapper [function, dict, or Series] Mapping correspondence.

na_action [{None, ‘ignore’}] If ‘ignore’, propagate NA values, without passing them
to the mapping correspondence.

Returns

applied [Union[Index, MultiIndex], inferred] The output of the mapping function ap-
plied to the index. If the function returns a tuple with more than one element a
MultiIndex will be returned.

pandas.Index.max

Index.max(axis=None, skipna=True, *args, **kwargs)
Return the maximum value of the Index.

Parameters

axis [int, optional] For compatibility with NumPy. Only 0 or None are allowed.

skipna [bool, default True] Exclude NA/null values when showing the result.

*args, **kwargs Additional arguments and keywords for compatibility with NumPy.

Returns

scalar Maximum value.

See also:

Index.min Return the minimum value in an Index.

Series.max Return the maximum value in a Series.

DataFrame.max Return the maximum values in a DataFrame.

Examples

```python
>>> idx = pd.Index([3, 2, 1])
>>> idx.max()
3

>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```python
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.max()
('b', 2)
```
pandas.Index.memory_usage

Index memory_usage(deep=False)
Memory usage of the values.

Parameters

deep [bool, default False] Introspect the data deeply, interrogate object dtypes for
system-level memory consumption.

Returns

bytes used

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of the array.

Notes

Memory usage does not include memory consumed by elements that are not components of the array if
deep=False or if used on PyPy

pandas.Index.min

Index min(axis=None, skipna=True, *args, **kwargs)
Return the minimum value of the Index.

Parameters

axis [{None}] Dummy argument for consistency with Series.
skipna [bool, default True] Exclude NA/null values when showing the result.
*args, **kwargs Additional arguments and keywords for compatibility with NumPy.

Returns

scalar Minimum value.

See also:

Index.max Return the maximum value of the object.
Series.min Return the minimum value in a Series.
DataFrame.min Return the minimum values in a DataFrame.

Examples

```python
>>> idx = pd.Index([3, 2, 1])
>>> idx.min()
1

>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```
For a MultiIndex, the minimum is determined lexicographically.

```python
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.min()
('a', 1)
```

**pandas.Index.notna**

**final** `Index.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

**Returns**

`numpy.ndarray[bool]` Boolean array to indicate which entries are not NA.

**See also:**

`Index.notnull` Alias of notna.

`Index.isna` Inverse of notna.

`notna` Top-level notna.

**Examples**

Show which entries in an Index are not NA. The result is an array.

```python
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True, True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```python
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True, True, True, False])
```
pandas.Index.notnull

Index.notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

numpy.ndarray[bool] Boolean array to indicate which entries are not NA.

See also:

Index.notnull Alias of notna.
Index.isna Inverse of notna.
notna Top-level notna.

Examples

Show which entries in an Index are not NA. The result is an array.

```python
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True, True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```python
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True, True, True, False])
```

pandas.Index.nunique

Index.nunique(dropna=True)

Return number of unique elements in the object.

Excludes NA values by default.

Parameters

- dropna [bool, default True] Don’t include NaN in the count.

Returns

- int

See also:
DataFrame.nunique  Method nunique for DataFrame.

Series.count  Count non-NA/null observations in the Series.

Examples

```python
>>> s = pd.Series([1, 3, 5, 7, 7])
>>> s
0 1
1 3
2 5
3 7
4 7
dtype: int64

>>> s.nunique()
4
```

pandas.Index.putmask

**final** Index.putmask(mask, value)

Return a new Index of the values set with the mask.

**Returns**

Index

**See also:**

numpy.ndarray.putmask  Changes elements of an array based on conditional and input values.

pandas.Index.ravel

**final** Index.ravel(order='C')

Return an ndarray of the flattened values of the underlying data.

**Returns**

numpy.ndarray  Flattened array.

**See also:**

numpy.ndarray.ravel  Return a flattened array.
Index.reindex(target, method=None, level=None, limit=None, tolerance=None)

Create index with target's values.

Parameters

target [an iterable]  
method [(None, ‘pad’/ffill’, ‘backfill’/bfill’, ‘nearest’), optional]
  • default: exact matches only.
  • pad / ffill: find the PREVIOUS index value if no exact match.
  • backfill / bfill: use NEXT index value if no exact match
  • nearest: use the NEAREST index value if no exact match. Tied distances are
    broken by preferring the larger index value.

level [int, optional]  Level of multiindex.

limit [int, optional]  Maximum number of consecutive labels in target to match for
  inexact matches.

tolerance [int or float, optional]  Maximum distance between original and new labels
  for inexact matches. The values of the index at the matching locations must satisfy
  the equation abs(index[indexer] - target) <= tolerance.

Tolerance may be a scalar value, which applies the same tolerance to all values, or
list-like, which applies variable tolerance per element. List-like includes list, tuple,
array, Series, and must be the same size as the index and its dtype must exactly
match the index’s type.

Returns

new_index [pd.Index]  Resulting index.

indexer [np.ndarray[np.intp] or None]  Indices of output values in original index.

Raises

TypeError  If method passed along with level.

ValueError  If non-unique multi-index

ValueError  If non-unique index and method or limit passed.

See also:

Series.reindex

DataFrame.reindex
3.6. Index objects

Examples

```python
>>> idx = pd.Index(['car', 'bike', 'train', 'tractor'])
>>> idx
Index(['car', 'bike', 'train', 'tractor'], dtype='O')
```

```python
>>> idx.reindex(['car', 'bike'])
(Index(['car', 'bike'], dtype='O'), array([0, 1]))
```

### pandas.Index.rename

**Index.rename(name, inplace=False)**

Alter Index or MultiIndex name.

Able to set new names without level. Defaults to returning new index. Length of names must match number of levels in MultiIndex.

**Parameters**

- `name` (label or list of labels) Name(s) to set.
- `inplace` (bool, default False) Modifies the object directly, instead of creating a new Index or MultiIndex.

**Returns**

Index or None The same type as the caller or None if `inplace=True`.

**See also:**

*Index.set_names* Able to set new names partially and by level.

Examples

```python
>>> idx = pd.Index(['A', 'C', 'A', 'B'], name='score')
>>> idx.rename('grade')
Index(['A', 'C', 'A', 'B'], dtype='O', name='grade')
```

```python
>>> idx.rename(['species', 'year'])
MultiIndex([('python', 2018),
            ('python', 2019),
            ('cobra', 2018),
            ('cobra', 2019),
            names=['kind', 'year'])
```

(continues on next page)
Traceback (most recent call last):
  TypeError: Must pass list-like as `names`.

pandas.Index.repeat

```
Index.repeat(repeats, axis=None)
```

Repeat elements of a Index.

Returns a new Index where each element of the current Index is repeated consecutively a given number of times.

- **Parameters**
  - `repeats` [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Index.
  - `axis` [None] Must be None. Has no effect but is accepted for compatibility with numpy.

- **Returns**
  - `repeated_index` [Index] Newly created Index with repeated elements.

- **See also:**
  - `Series.repeat` Equivalent function for Series.
  - `numpy.repeat` Similar method for `numpy.ndarray`.

- **Examples**

```
>>> idx = pd.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.repeat(2)
Index(['a', 'a', 'b', 'b', 'c', 'c'], dtype='object')
>>> idx.repeat([1, 2, 3])
Index(['a', 'b', 'b', 'c', 'c', 'c'], dtype='object')
```

pandas.Index.searchsorted

```
Index.searchsorted(value, side='left', sorter=None)
```

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Index `self` such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

- **Note:** The `Index` must be monotonically sorted, otherwise wrong locations will likely be returned. Pandas does not check this for you.

- **Parameters**
  - `value` [array-like or scalar] Values to insert into `self`. 
side  [['left', 'right'], optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of self).

sorter  [1-D array-like, optional] Optional array of integer indices that sort self into ascending order. They are typically the result of np.argsort.

Returns

int or array of int  A scalar or array of insertion points with the same shape as value.

See also:

sort_values  Sort by the values along either axis.

numpy.searchsorted  Similar method from NumPy.

Notes

Binary search is used to find the required insertion points.

Examples

```python
>>> ser = pd.Series([1, 2, 3])
>>> ser
0    1
1    2
2    3
dtype: int64

>>> ser.searchsorted(4)
3

>>> ser.searchsorted([0, 4])
array([0, 3])

>>> ser.searchsorted([1, 3], side='left')
array([0, 2])

>>> ser.searchsorted([1, 3], side='right')
array([1, 3])
```

```python
>>> ser
0    2000-03-11
1    2000-03-12
2    2000-03-13
dtype: datetime64[ns]

>>> ser.searchsorted('3/14/2000')
3
```
>>> ser = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk'], ordered=True)

If the values are not monotonically sorted, wrong locations may be returned:

```python
>>> ser = pd.Series([2, 1, 3])
>>> ser
0 2
1 1
2 3
dtype: int64
```

```python
>>> ser.searchsorted(1)
0 # wrong result, correct would be 1
```

### pandas.Index.set_names

`Index.set_names(names, level=None, inplace=False)`

Set Index or MultiIndex name.

Able to set new names partially and by level.

**Parameters**

- **names** [label or list of label or dict-like for MultiIndex] Name(s) to set.
  
  Changed in version 1.3.0.

- **level** [int, label or list of int or label, optional] If the index is a MultiIndex and names is not dict-like, level(s) to set (None for all levels). Otherwise level must be None.
  
  Changed in version 1.3.0.

- **inplace** [bool, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

**Returns**

- **Index or None** The same type as the caller or None if `inplace=True`.

**See also:**

- **Index.rename** Able to set new names without level.
Examples

```python
>>> idx = pd.Index([1, 2, 3, 4])
>>> idx
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx.set_names('quarter')
Int64Index([1, 2, 3, 4], dtype='int64', name='quarter')

>>> idx = pd.MultiIndex.from_product([['python', 'cobra'], [2018, 2019]])
>>> idx
MultiIndex([('python', 2018),
             ('python', 2019),
             ('cobra', 2018),
             ('cobra', 2019)],
            names=['kind', 'year'])
>>> idx.set_names(['kind', 'year'], inplace=True)
>>> idx
MultiIndex([('python', 2018),
             ('python', 2019),
             ('cobra', 2018),
             ('cobra', 2019)],
            names=['kind', 'year'])

When renaming levels with a dict, levels can not be passed.

>>> idx.set_names({'kind': 'snake'})
MultiIndex([('python', 2018),
             ('python', 2019),
             ('cobra', 2018),
             ('cobra', 2019)],
            names=['snake', 'year'])
```
**pandas**.Index.set_value

**final** Index.set_value(arr, key, value)

Fast lookup of value from 1-dimensional ndarray.

Deprecated since version 1.0.

**Notes**

Only use this if you know what you’re doing.

**pandas**.Index.shift

Index.shift(periods=1, freq=None)

Shift index by desired number of time frequency increments.

This method is for shifting the values of datetime-like indexes by a specified time increment a given number of times.

**Parameters**

- **periods** [int, default 1] Number of periods (or increments) to shift by, can be positive or negative.
- **freq** [pandas.DateOffset, pandas.Timedelta or str, optional] Frequency increment to shift by. If None, the index is shifted by its own freq attribute. Offset aliases are valid strings, e.g., ‘D’, ‘W’, ‘M’ etc.

**Returns**

pandas.Index  Shifted index.

See also:

Series.shift  Shift values of Series.

**Notes**

This method is only implemented for datetime-like index classes, i.e., DatetimeIndex, PeriodIndex and TimedeltaIndex.

**Examples**

Put the first 5 month starts of 2011 into an index.

```python
>>> month_starts = pd.date_range('1/1/2011', periods=5, freq='MS')
>>> month_starts
DatetimeIndex(['2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
                '2011-05-01'],
               dtype='datetime64[ns]', freq='MS')
```

Shift the index by 10 days.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> month_starts.shift(10, freq='D')
              '2011-05-11'],
          dtype='datetime64[ns]', freq=None)
```

The default value of `freq` is the `freq` attribute of the index, which is ‘MS’ (month start) in this example.

```python
>>> month_starts.shift(10)
DatetimeIndex(['2011-11-01', '2011-12-01', '2012-01-01', '2012-02-01',
              '2012-03-01'],
          dtype='datetime64[ns]', freq='MS')
```

**pandas.Index.slice_indexer**

`Index.slice_indexer(start=None, end=None, step=None, kind=NoDefault.no_default)`

Compute the slice indexer for input labels and step.

Index needs to be ordered and unique.

**Parameters**

- **start** [label, default None] If None, defaults to the beginning.
- **end** [label, default None] If None, defaults to the end.
- **step** [int, default None]
- **kind** [str, default None] Deprecated since version 1.4.0.

**Returns**

- **indexer** [slice]

**Raises**

- **KeyError** [If key does not exist, or key is not unique and index is] not ordered.

**Notes**

This function assumes that the data is sorted, so use at your own peril

**Examples**

This is a method on all index types. For example you can do:

```python
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_indexer(start='b', end='c')
slice(1, 3, None)
```

```python
>>> idx = pd.MultiIndex.from_arrays([[list('abcd'), list('efgh')]])
>>> idx.slice_indexer(start='b', end=('c', 'g'))
slice(1, 3, None)
```
pandas.Index.slice_locs

`Index.slice_locs(start=None, end=None, step=None, kind=None)`

Compute slice locations for input labels.

**Parameters**

- **start** [label, default None] If None, defaults to the beginning.
- **end** [label, default None] If None, defaults to the end.
- **step** [int, defaults None] If None, defaults to 1.
- **kind** [{'loc', 'getitem'} or None] Deprecated since version 1.4.0.

**Returns**

- **start, end** [int]

**See also:**

- `Index.get_loc` Get location for a single label.

**Notes**

This method only works if the index is monotonic or unique.

**Examples**

```python
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

pandas.Index.sort

`final Index.sort(*args, **kwargs)`

Use sort_values instead.

pandas.Index.sort_values

`Index.sort_values(return_indexer=False, ascending=True, na_position='last', key=None)`

Return a sorted copy of the index.

Return a sorted copy of the index, and optionally return the indices that sorted the index itself.

**Parameters**

- **return_indexer** [bool, default False] Should the indices that would sort the index be returned.
- **ascending** [bool, default True] Should the index values be sorted in an ascending order.
- **na_position** [{'first' or 'last'}, default 'last'] Argument ‘first’ puts NaNs at the beginning, ‘last’ puts NaNs at the end.

New in version 1.2.0.
key [callable, optional] If not None, apply the key function to the index values before sorting. This is similar to the key argument in the builtin sorted() function, with the notable difference that this key function should be vectorized. It should expect an Index and return an Index of the same shape.

New in version 1.1.0.

Returns

- sorted_index [pandas.Index] Sorted copy of the index.
- indexer [numpy.ndarray, optional] The indices that the index itself was sorted by.

See also:

- Series.sort_values Sort values of a Series.
- DataFrame.sort_values Sort values in a DataFrame.

Examples

```python
>>> idx = pd.Index([10, 100, 1, 1000])
```

Sort values in ascending order (default behavior).

```python
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order, and also get the indices `idx` was sorted by.

```python
>>> idx.sort_values(ascending=False, return_indexer=True)
(Int64Index([1000, 100, 10, 1], dtype='int64'), array([3, 1, 0, 2]))
```

pandas.Index.sortlevel

Index.sortlevel(level=None, ascending=True, sort_remaining=None)

For internal compatibility with the Index API.

Sort the Index. This is for compat with MultiIndex

Parameters

- ascending [bool, default True] False to sort in descending order

- level, sort_remaining are compat parameters

Returns

- Index
pandas.Index.str

Index.str()

Vectorized string functions for Series and Index.
NAs stay NA unless handled otherwise by a particular method. Patterned after Python’s string methods, with some inspiration from R’s stringr package.

Examples

```python
>>> s = pd.Series(['A_Str_Series'])
>>> s
0   A_Str_Series
dtype: object

>>> s.str.split('_')
0   [A, Str, Series]
dtype: object

>>> s.str.replace('_', '')
0   AStrSeries
```

pandas.Index.symmetric_difference

Index.symmetric_difference(other, result_name=None, sort=None)

Compute the symmetric difference of two Index objects.

Parameters

- **other** [Index or array-like]
- **result_name** [str]
- **sort** [False or None, default None] Whether to sort the resulting index. By default, the values are attempted to be sorted, but any TypeError from incomparable elements is caught by pandas.
  - None: Attempt to sort the result, but catch any TypeErrors from comparing incomparable elements.
  - False: Do not sort the result.

Returns

- **symmetric_difference** [Index]
Notes

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

Examples

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

**pandas.Index.take**

`Index.take(indices, axis=0, allow_fill=True, fill_value=None, **kwargs)`  
Return a new Index of the values selected by the indices.

For internal compatibility with numpy arrays.

**Parameters**

- `indices` [array-like] Indices to be taken.
- `axis` [int, optional] The axis over which to select values, always 0.
- `allow_fill` [bool, default True]
- `fill_value` [scalar, default None] If `allow_fill=True` and `fill_value` is not None, indices specified by -1 are regarded as NA. If Index doesn’t hold NA, raise ValueError.

**Returns**

- `Index` An index formed of elements at the given indices. Will be the same type as self, except for RangeIndex.

**See also:**

- `numpy.ndarray.take` Return an array formed from the elements of a at the given indices.

**pandas.Index.to_flat_index**

`Index.to_flat_index()`  
Identity method.

This is implemented for compatibility with subclass implementations when chaining.

**Returns**

- `pd.Index` Caller.

**See also:**

- `MultiIndex.to_flat_index` Subclass implementation.
Index to_frame

Index.to_frame(index=True, name=NoDefault.no_default)

Create a DataFrame with a column containing the Index.

Parameters

index  [bool, default True] Set the index of the returned DataFrame as the original Index.

name  [object, default None] The passed name should substitute for the index name (if it has one).

Returns

DataFrame  DataFrame containing the original Index data.

See also:

Index.to_series  Convert an Index to a Series.

Series.to_frame  Convert Series to DataFrame.

Examples

```python
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
animal
Ant    Ant
Bear   Bear
Cow    Cow
```

By default, the original Index is reused. To enforce a new Index:

```python
>>> idx.to_frame(index=False)
animal
0   Ant
1   Bear
2   Cow
```

To override the name of the resulting column, specify name:

```python
>>> idx.to_frame(index=False, name='zoo')
zoo
0   Ant
1   Bear
2   Cow
```
pandas.Index.to_list

Index.to_list()

Return a list of the values.
These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

Returns
list

See also:

numpy.ndarray.tolist Return the array as an a.ndim-levels deep nested list of Python scalars.

pandas.Index.to_native_types

final Index.to_native_types(slicer=None, **kwargs)

Format specified values of self and return them.
Deprecated since version 1.2.0.

Parameters

slicer [int, array-like] An indexer into self that specifies which values are used in the formatting process.

kwangs [dict] Options for specifying how the values should be formatted. These options include the following:
1) na_rep [str] The value that serves as a placeholder for NULL values
2) quoting [bool or None] Whether or not there are quoted values in self
3) date_format [str] The format used to represent date-like values.

Returns

numpy.ndarray Formatted values.

pandas.Index.to_numpy

Index.to_numpy(dtype=None, copy=False, na_value=NoDefault.no_default, **kwargs)

A NumPy ndarray representing the values in this Series or Index.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to numpy.asarray().

copy [bool, default False] Whether to ensure that the returned value is not a view on another array. Note that copy=False does not ensure that to_numpy() is no-copy. Rather, copy=True ensure that a copy is made, even if not strictly necessary.

na_value [Any, optional] The value to use for missing values. The default value depends on dtype and the type of the array.
New in version 1.0.0.
**kws** Additional keywords passed through to the to_numpy method of the underlying array (for extension arrays).

New in version 1.0.0.

**Returns**

numpy.ndarray

**See also:**

Series.array Get the actual data stored within.

Index.array Get the actual data stored within.

DataFrame.to_numpy Similar method for DataFrame.

**Notes**

The returned array will be the same up to equality (values equal in self will be equal in the returned array; likewise for values that are not equal). When self contains an ExtensionArray, the dtype may be different. For example, for a category-dtype Series, to_numpy() will return a NumPy array and the categorical dtype will be lost.

For NumPy dtypes, this will be a reference to the actual data stored in this Series or Index (assuming copy=False). Modifying the result in place will modify the data stored in the Series or Index (not that we recommend doing that).

For extension types, to_numpy() may require copying data and coercing the result to a NumPy type (possibly object), which may be expensive. When you need a no-copy reference to the underlying data, Series.array should be used instead.

This table lays out the different dtypes and default return types of to_numpy() for various dtypes within pandas.

<table>
<thead>
<tr>
<th>dtype</th>
<th>array type</th>
</tr>
</thead>
<tbody>
<tr>
<td>category[T]</td>
<td>ndarray[T] (same dtype as input)</td>
</tr>
<tr>
<td>period</td>
<td>ndarray[object] (Periods)</td>
</tr>
<tr>
<td>interval</td>
<td>ndarray[object] (Intervals)</td>
</tr>
<tr>
<td>IntegerNA</td>
<td>ndarray[object]</td>
</tr>
<tr>
<td>datetime64[ns]</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>datetime64[ns, tz]</td>
<td>ndarray[object] (Timestamps)</td>
</tr>
</tbody>
</table>

**Examples**

```python
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

Specify the dtype to control how datetime-aware data is represented. Use dtype=object to return an ndarray of pandas Timestamp objects, each with the correct tz.

```python
>>> ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> ser.to_numpy(dtype=object)
array([Timestamp('2000-01-01 00:00:00+0100'), tz='CET'],
```

(continues on next page)
Or dtype='datetime64[ns]' to return an ndarray of native datetime64 values. The values are converted to UTC and the timezone info is dropped.

```python
>>> ser.to_numpy(dtype="datetime64[ns]"")
... array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00...'],
dtype='datetime64[ns]')
```

### pandas.Index.to_series

**Index.to_series** *(index=None, name=None)*

Create a Series with both index and values equal to the index keys.

Useful with `map` for returning an indexer based on an index.

**Parameters**

- **index**  [Index, optional] Index of resulting Series. If None, defaults to original index.
- **name**  [str, optional] Name of resulting Series. If None, defaults to name of original index.

**Returns**

- **Series**  The dtype will be based on the type of the Index values.

**See also:**

- **Index.to_frame**  Convert an Index to a DataFrame.
- **Series.to_frame**  Convert Series to DataFrame.

**Examples**

```python
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
```

By default, the original Index and original name is reused.

```python
>>> idx.to_series()
animal
Ant    Ant
Bear   Bear
Cow    Cow
Name: animal, dtype: object
```

To enforce a new Index, specify new labels to `index`:

```python
>>> idx.to_series(index=[0, 1, 2])
0    Ant
1    Bear
Name: animal, dtype: object
```

(continues on next page)
To override the name of the resulting column, specify name:

```python
>>> idx.to_series(name='zoo')
animal
Ant    Ant
Bear   Bear
Cow    Cow
Name: zoo, dtype: object
```

### pandas.Index.tolist

Index.tolist()

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

**Returns**

list

**See also:**

numpy.ndarray.tolist Return the array as an `a.ndim`-levels deep nested list of Python scalars.

### pandas.Index.transpose

Index.transpose(*args, **kwargs)

Return the transpose, which is by definition self.

**Returns**

%(klass)s

### pandas.Index.union

final Index.union(other, sort=None)

Form the union of two Index objects.

If the Index objects are incompatible, both Index objects will be cast to dtype('object') first.

Changed in version 0.25.0.

**Parameters**

other [Index or array-like]

sort [bool or None, default None] Whether to sort the resulting Index.

- None : Sort the result, except when
  1. `self` and `other` are equal.
2. `self` or `other` has length 0.

3. Some values in `self` or `other` cannot be compared. A RuntimeWarning is issued in this case.
   - False : do not sort the result.

Returns

union [Index]

Examples

Union matching dtypes

```python
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.union(idx2)
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

Union mismatched dtypes

```python
>>> idx1 = pd.Index(['a', 'b', 'c', 'd'])
>>> idx2 = pd.Index([1, 2, 3, 4])
>>> idx1.union(idx2)
Index(['a', 'b', 'c', 'd', 1, 2, 3, 4], dtype='object')
```

MultiIndex case

```python
>>> idx1 = pd.MultiIndex.from_arrays(
...     [[1, 1, 2, 2], ['Red', 'Blue', 'Red', 'Blue']]
... )
>>> idx1
MultiIndex([(1, 'Red'),
(1, 'Blue'),
(2, 'Red'),
(2, 'Blue')],
)  
>>> idx2 = pd.MultiIndex.from_arrays(
...     [[3, 3, 2, 2], ['Red', 'Green', 'Red', 'Green']]
... )
>>> idx2
MultiIndex([(3, 'Red'),
(3, 'Green'),
(2, 'Red'),
(2, 'Green')],
)  
>>> idx1.union(idx2)
MultiIndex([(1, 'Blue'),
(1, 'Red'),
(2, 'Blue'),
(2, 'Green'),
(2, 'Red'),
(3, 'Green'),
(3, 'Red')],
)...)
```
>>> idx1.union(idx2, sort=False)
MultiIndex([(1, 'Red'),
            (1, 'Blue'),
            (2, 'Red'),
            (2, 'Blue'),
            (3, 'Red'),
            (3, 'Green'),
            (2, 'Green')],
           )

**pandas.Index.unique**

Index.unique(*level=None*)

Return unique values in the index.

Unique values are returned in order of appearance, this does NOT sort.

**Parameters**

- **level** [int or hashable, optional] Only return values from specified level (for MultiIndex). If int, gets the level by integer position, else by level name.

**Returns**

Index

**See also:**

unique Numpy array of unique values in that column.

Series.unique Return unique values of Series object.

**pandas.Index.value_counts**

Index.value_counts(*normalize=False, sort=True, ascending=False, bins=None, dropna=True*)

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

**Parameters**

- **normalize** [bool, default False] If True then the object returned will contain the relative frequencies of the unique values.
- **sort** [bool, default True] Sort by frequencies.
- **ascending** [bool, default False] Sort in ascending order.
- **bins** [int, optional] Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data.
- **dropna** [bool, default True] Don’t include counts of NaN.

**Returns**

Series
See also:

- **Series.count**  Number of non-NA elements in a Series.
- **DataFrame.count**  Number of non-NA elements in a DataFrame.
- **DataFrame.value_counts**  Equivalent method on DataFrames.

**Examples**

```python
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0  2
1.0  1
2.0  1
4.0  1
dtype: int64
```

With `normalize` set to `True`, returns the relative frequency by dividing all values by the sum of values.

```python
>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
>>> s.value_counts(normalize=True)
3.0  0.4
1.0  0.2
2.0  0.2
4.0  0.2
dtype: float64
```

**bins**

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

```python
>>> s.value_counts(bins=3)
(0.996, 2.0]  2
(2.0, 3.0]   2
(3.0, 4.0]   1
dtype: int64
```

**dropna**

With `dropna` set to `False` we can also see NaN index values.

```python
>>> s.value_counts(dropna=False)
3.0  2
1.0  1
2.0  1
4.0  1
NaN  1
dtype: int64
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.Index.where**

```python
final Index.where(cond, other=None)
```

Replace values where the condition is False.
The replacement is taken from other.

**Parameters**

- **cond** [bool array-like with the same length as self] Condition to select the values on.
- **other** [scalar, or array-like, default None] Replacement if the condition is False.

**Returns**

- **pandas.Index** A copy of self with values replaced from other where the condition is False.

**See also:**

- **Series.where** Same method for Series.
- **DataFrame.where** Same method for DataFrame.

**Examples**

```python
>>> idx = pd.Index(['car', 'bike', 'train', 'tractor'])
>>> idx
Index(['car', 'bike', 'train', 'tractor'], dtype='O')
>>> idx.isin(['car', 'train'])
Index(['car', 'other', 'train', 'other'], dtype='O')
```
Properties

- **Index.values**: Return an array representing the data in the Index.
- **Index.is_monotonic**: Alias for `is_monotonic_increasing`.
- **Index.is_monotonic_increasing**: Return if the index is monotonic increasing (only equal or increasing) values.
- **Index.is_monotonic_decreasing**: Return if the index is monotonic decreasing (only equal or decreasing) values.
- **Index.is_unique**: Return if the index has unique values.
- **Index.has_duplicates**: Check if the Index has duplicate values.
- **Index.hasnans**: Return True if there are any NaNs.
- **Index.dtype**: Return the dtype object of the underlying data.
- **Index.inferred_type**: Return a string of the type inferred from the values.
- **Index.is_all_dates**: Whether or not the index values only consist of dates.
- **Index.shape**: Return a tuple of the shape of the underlying data.
- **Index.name**: Return Index or MultiIndex name.
- **Index.names**: Return a list of names if the index is a MultiIndex.
- **Index.nbytes**: Return the number of bytes in the underlying data.
- **Index.ndim**: Number of dimensions of the underlying data, by definition 1.
- **Index.size**: Return the number of elements in the underlying data.
- **Index.empty**: Return the transpose, which is by definition self.
- **Index.memory_usage([deep])**: Memory usage of the values.

**pandas.Index.names**

**property** `Index.names`

**pandas.Index.empty**

**property** `Index.empty`

Modifying and computations

- **Index.all(**args, **kwargs)**: Return whether all elements are Truthy.
- **Index.any(**args, **kwargs)**: Return whether any element is Truthy.
- **Index.argmin([axis, skipna])**: Return int position of the smallest value in the Series.
- **Index.argmax([axis, skipna])**: Return int position of the largest value in the Series.
- **Index.copy([name, deep, dtype, names])**: Make a copy of this object.
- **Index.delete(loc)**: Make new Index with passed location(-s) deleted.
- **Index.drop(labels[, errors])**: Make new Index with passed list of labels deleted.
- **Index.drop_duplicates([keep])**: Return Index with duplicate values removed.
- **Index.duplicated([keep])**: Indicate duplicate index values.
- **Index.equals(other)**: Determine if two Index object are equal.
### Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.factorize()</code></td>
<td>Encode the object as an enumerated type or categorical variable.</td>
</tr>
<tr>
<td><code>Index.identical()</code></td>
<td>Similar to equals, but checks that object attributes and types are also equal.</td>
</tr>
<tr>
<td><code>Index.insert()</code></td>
<td>Make new Index inserting new item at location.</td>
</tr>
<tr>
<td><code>Index.is_()</code></td>
<td>More flexible, faster check like <code>is</code> but that works through views.</td>
</tr>
<tr>
<td><code>Index.is_boolean()</code></td>
<td>Check if the Index only consists of booleans.</td>
</tr>
<tr>
<td><code>Index.is_categorical()</code></td>
<td>Check if the Index holds categorical data.</td>
</tr>
<tr>
<td><code>Index.is_float()</code></td>
<td>Check if the Index is a floating type.</td>
</tr>
<tr>
<td><code>Index.is_int()</code></td>
<td>Check if the Index only consists of integers.</td>
</tr>
<tr>
<td><code>Index.is_interval()</code></td>
<td>Check if the Index holds Interval objects.</td>
</tr>
<tr>
<td><code>Index.is_mixed()</code></td>
<td>Check if the Index holds data with mixed data types.</td>
</tr>
<tr>
<td><code>Index.is_numeric()</code></td>
<td>Check if the Index only consists of numeric data.</td>
</tr>
<tr>
<td><code>Index.min()</code></td>
<td>Return the minimum value of the Index.</td>
</tr>
<tr>
<td><code>Index.max()</code></td>
<td>Return the maximum value of the Index.</td>
</tr>
<tr>
<td><code>Index.reindex()</code></td>
<td>Create index with target's values.</td>
</tr>
<tr>
<td><code>Index.rename()</code></td>
<td>Alter Index or MultiIndex name.</td>
</tr>
<tr>
<td><code>Index.repeat()</code></td>
<td>Repeat elements of a Index.</td>
</tr>
<tr>
<td><code>Index.where()</code></td>
<td>Replace values where the condition is False.</td>
</tr>
<tr>
<td><code>Index.take()</code></td>
<td>Return a new Index of the values selected by the indices.</td>
</tr>
<tr>
<td><code>Index.putmask()</code></td>
<td>Return a new Index of the values set with the mask.</td>
</tr>
<tr>
<td><code>Index.unique()</code></td>
<td>Return unique values in the index.</td>
</tr>
<tr>
<td><code>Index.nunique()</code></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><code>Index.value_counts()</code></td>
<td>Return a Series containing counts of unique values.</td>
</tr>
</tbody>
</table>

#### Compatibility with MultiIndex

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.set_names()</code></td>
<td>Set Index or MultiIndex name.</td>
</tr>
<tr>
<td><code>Index.droplevel()</code></td>
<td>Return index with requested level(s) removed.</td>
</tr>
</tbody>
</table>

#### Missing values

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.fillna()</code></td>
<td>Fill NA/NaN values with the specified value.</td>
</tr>
<tr>
<td><code>Index.dropna()</code></td>
<td>Return Index without NA/NaN values.</td>
</tr>
<tr>
<td><code>Index.isna()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>Index.notna()</code></td>
<td>Detect existing (non-missing) values.</td>
</tr>
</tbody>
</table>
Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.astype(dtype[, copy])</code></td>
<td>Create an Index with values cast to dtypes.</td>
</tr>
<tr>
<td><code>Index.item()</code></td>
<td>Return the first element of the underlying data as a Python scalar.</td>
</tr>
<tr>
<td><code>Index.map(mapper[, na_action])</code></td>
<td>Map values using an input mapping or function.</td>
</tr>
<tr>
<td><code>Index.ravel([order])</code></td>
<td>Return an ndarray of the flattened values of the underlying data.</td>
</tr>
<tr>
<td><code>Index.to_list()</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>Index.to_native_types([slicer])</code></td>
<td>(DEPRECATED) Format specified values of self and return them.</td>
</tr>
<tr>
<td><code>Index.to_series([index, name])</code></td>
<td>Create a Series with both index and values equal to the index keys.</td>
</tr>
<tr>
<td><code>Index.to_frame([index, name])</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td><code>Index.view([cls])</code></td>
<td></td>
</tr>
</tbody>
</table>

**pandas.Index.view**

`Index.view(cls=None)`

Sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.argsort(*args, **kwargs)</code></td>
<td>Return the integer indices that would sort the index.</td>
</tr>
<tr>
<td><code>Index.searchsorted(value[, side, sorter])</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>Index.sort_values([return_indexer, ...])</code></td>
<td>Return a sorted copy of the index.</td>
</tr>
</tbody>
</table>

Time-specific operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.shift([periods, freq])</code></td>
<td>Shift index by desired number of time frequency increments.</td>
</tr>
</tbody>
</table>

Combining / joining / set operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.append(other)</code></td>
<td>Append a collection of Index options together.</td>
</tr>
<tr>
<td><code>Index.join(other[, how, level, ...])</code></td>
<td>Compute join_index and indexers to conform data structures to the new index.</td>
</tr>
<tr>
<td><code>Index.intersection(other[, sort])</code></td>
<td>Form the intersection of two Index objects.</td>
</tr>
<tr>
<td><code>Index.union(other[, sort])</code></td>
<td>Form the union of two Index objects.</td>
</tr>
<tr>
<td><code>Index.difference(other[, sort])</code></td>
<td>Return a new Index with elements of index not in other.</td>
</tr>
<tr>
<td><code>Index.symmetric_difference(other[, ...])</code></td>
<td>Compute the symmetric difference of two Index objects.</td>
</tr>
</tbody>
</table>

3.6. Index objects
### Selecting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Index.asof(label)</code></td>
<td>Return the label from the index, or, if not present, the previous one.</td>
</tr>
<tr>
<td><code>Index.asof_locs(where, mask)</code></td>
<td>Return the locations (indices) of labels in the index.</td>
</tr>
<tr>
<td><code>Index.get_indexer(target[, method, limit, ...])</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>Index.get_indexer_for(target)</code></td>
<td>Guaranteed return of an indexer even when non-unique.</td>
</tr>
<tr>
<td><code>Index.get_indexer_non_unique(target)</code></td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td><code>Index.get_level_values(level)</code></td>
<td>Return an Index of values for requested level.</td>
</tr>
<tr>
<td><code>Index.get_loc(key[, method, tolerance])</code></td>
<td>Get integer location, slice or boolean mask for requested label.</td>
</tr>
<tr>
<td><code>Index.get_slice_bound(label, side[, kind])</code></td>
<td>Calculate slice bound that corresponds to given label.</td>
</tr>
<tr>
<td><code>Index.isin(values[, level])</code></td>
<td>Return a boolean array where the index values are in <code>values</code>.</td>
</tr>
<tr>
<td><code>Index.slice_indexer([start, end, step, kind])</code></td>
<td>Compute the slice indexer for input labels and step.</td>
</tr>
<tr>
<td><code>Index.slice_locs([start, end, step, kind])</code></td>
<td>Compute slice locations for input labels.</td>
</tr>
</tbody>
</table>

### 3.6.2 Numeric Index

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RangeIndex([]start, stop, step, dtype, copy, ...])</td>
<td>Immutable Index implementing a monotonic integer range.</td>
</tr>
<tr>
<td>Int64Index([data, dtype, copy, name])</td>
<td>(DEPRECATED) Immutable sequence used for indexing and alignment.</td>
</tr>
<tr>
<td>UInt64Index([data, dtype, copy, name])</td>
<td>(DEPRECATED) Immutable sequence used for indexing and alignment.</td>
</tr>
<tr>
<td>Float64Index([data, dtype, copy, name])</td>
<td>(DEPRECATED) Immutable sequence used for indexing and alignment.</td>
</tr>
</tbody>
</table>

**pandas.RangeIndex**

```python
class pandas.RangeIndex(start=None, stop=None, step=None, dtype=None, copy=False, name=None)
```

Immutable Index implementing a monotonic integer range.

RangeIndex is a memory-saving special case of Int64Index limited to representing monotonic ranges. Using RangeIndex may in some instances improve computing speed.

This is the default index type used by DataFrame and Series when no explicit index is provided by the user.

**Parameters**

- **start** [int (default: 0), range, or other RangeIndex instance] If `int` and “stop” is not given, interpreted as “stop” instead.
- **stop** [int (default: 0)]
- **step** [int (default: 1)]
- **dtype** [np.int64] Unused, accepted for homogeneity with other index types.
- **copy** [bool, default False] Unused, accepted for homogeneity with other index types.
- **name** [object, optional] Name to be stored in the index.
See also:

- **Index**  The base pandas Index type.
- **Int64Index**  Index of int64 data.

### Attributes

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>The value of the <em>start</em> parameter (0 if this was not supplied).</td>
</tr>
<tr>
<td>stop</td>
<td>The value of the <em>stop</em> parameter.</td>
</tr>
<tr>
<td>step</td>
<td>The value of the <em>step</em> parameter (1 if this was not supplied).</td>
</tr>
</tbody>
</table>

```
pandas.RangeIndex.start
```

**property**  `RangeIndex.start`

The value of the *start* parameter (0 if this was not supplied).

```
pandas.RangeIndex.stop
```

**property**  `RangeIndex.stop`

The value of the *stop* parameter.

```
pandas.RangeIndex.step
```

**property**  `RangeIndex.step`

The value of the *step* parameter (1 if this was not supplied).

### Methods

```
from_range(data[, name, dtype])
```

Create RangeIndex from a range object.

```
pandas.RangeIndex.from_range
```

**classmethod**  `RangeIndex.from_range(data, name=None, dtype=None)`  
Create RangeIndex from a range object.

#### Returns

- **RangeIndex**
**pandas.Int64Index**

**class pandas.Int64Index(data=None, dtype=None, copy=False, name=None)**

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects. Int64Index is a special case of Index with purely integer labels.

Deprecated since version 1.4.0: In pandas v2.0 Int64Index will be removed and NumericIndex used instead. Int64Index will remain fully functional for the duration of pandas 1.x.

**Parameters**

- **data** [array-like (1-dimensional)]
- **dtype** [NumPy dtype (default: int64)]
- **copy** [bool] Make a copy of input ndarray.
- **name** [object] Name to be stored in the index.

**See also:**

- **Index** The base pandas Index type.
- **NumericIndex** Index of numpy int/uint/float data.

**Notes**

An Index instance can only contain hashable objects.

**Attributes**

- None

**Methods**

- None

**pandas.UInt64Index**

**class pandas.UInt64Index(data=None, dtype=None, copy=False, name=None)**

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects. UInt64Index is a special case of Index with purely unsigned integer labels.

Deprecated since version 1.4.0: In pandas v2.0 UInt64Index will be removed and NumericIndex used instead. UInt64Index will remain fully functional for the duration of pandas 1.x.

**Parameters**

- **data** [array-like (1-dimensional)]
- **dtype** [NumPy dtype (default: uint64)]
- **copy** [bool] Make a copy of input ndarray.
- **name** [object] Name to be stored in the index.

**See also:**

- **Index** The base pandas Index type.
- **NumericIndex** Index of numpy int/uint/float data.
Notes

An Index instance can only contain hashable objects.

Attributes

None

Methods

None

pandas.Float64Index

class pandas.Float64Index(data=None, dtype=None, copy=False, name=None)

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects. Float64Index is a special case of Index with purely float labels.

Deprecated since version 1.4.0: In pandas v2.0 Float64Index will be removed and NumericIndex used instead. Float64Index will remain fully functional for the duration of pandas 1.x.

Parameters

data [array-like (1-dimensional)]
dtype [NumPy dtype (default: float64)]
copy [bool] Make a copy of input ndarray.
name [object] Name to be stored in the index.

See also:

Index The base pandas Index type.
NumericIndex Index of numpy int/uint/float data.

Notes

An Index instance can only contain hashable objects.

Attributes

None
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RangeIndex.start</code></td>
<td>The value of the <code>start</code> parameter (0 if this was not supplied).</td>
</tr>
<tr>
<td><code>RangeIndex.stop</code></td>
<td>The value of the <code>stop</code> parameter.</td>
</tr>
<tr>
<td><code>RangeIndex.step</code></td>
<td>The value of the <code>step</code> parameter (1 if this was not supplied).</td>
</tr>
<tr>
<td><code>RangeIndex.from_range</code></td>
<td>Create <code>RangeIndex</code> from a range object.</td>
</tr>
</tbody>
</table>

3.6.3 CategoricalIndex

`CategoricalIndex([data, categories, ...])` Index based on an underlying `Categorical`.

class pandas.CategoricalIndex(data=None, categories=None, ordered=None, dtype=None, copy=False, name=None)

Index based on an underlying `Categorical`.

CategoricalIndex, like Categorical, can only take on a limited, and usually fixed, number of possible values (`categories`). Also, like Categorical, it might have an order, but numerical operations (additions, divisions, ...) are not possible.

Parameters

- `data` [array-like (1-dimensional)] The values of the categorical. If `categories` are given, values not in `categories` will be replaced with NaN.
- `categories` [index-like, optional] The categories for the categorical. Items need to be unique.
  If the categories are not given here (and also not in `dtype`), they will be inferred from the `data`.
- `ordered` [bool, optional] Whether or not this categorical is treated as an ordered categorical.
  If not given here or in `dtype`, the resulting categorical will be unordered.
- `dtype` [CategoricalDtype or “category”, optional] If `CategoricalDtype`, cannot be used together with `categories` or `ordered`.
- `copy` [bool, default False] Make a copy of input ndarray.
- `name` [object, optional] Name to be stored in the index.

Raises

- `ValueError` If the categories do not validate.
- `TypeError` If an explicit ordered=True is given but no categories and the values are not sortable.

See also:

- `Index` The base pandas Index type.
- `Categorical` A categorical array.
- `CategoricalDtype` Type for categorical data.
Notes

See the user guide for more.

Examples

```python
>>> pd.CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'])
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

CategoricalIndex can also be instantiated from a Categorical:

```python
>>> c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
>>> pd.CategoricalIndex(c)
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

Ordered CategoricalIndex can have a min and max value.

```python
>>> ci = pd.CategoricalIndex(
... ['a', 'b', 'c', 'a', 'b', 'c'], ordered=True, categories=['c', 'b', 'a']
... )
>>> ci
CategoricalIndex(['a', 'b', 'c'], categories=['c', 'b', 'a'], ordered=True, dtype='category')
>>> ci.min()
'c'
```

Attributes

<table>
<thead>
<tr>
<th>codes</th>
<th>The category codes of this categorical.</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories</td>
<td>The categories of this categorical.</td>
</tr>
<tr>
<td>ordered</td>
<td>Whether the categories have an ordered relationship.</td>
</tr>
</tbody>
</table>

```python
pandas.CategoricalIndex.codes
```

property `CategoricalIndex.codes`

The category codes of this categorical.

Codes are an array of integers which are the positions of the actual values in the categories array.

There is no setter, use the other categorical methods and the normal item setter to change values in the categorical.

Returns

* ndarray[int] A non-writable view of the `codes` array.

3.6. Index objects
pandas.CategoricalIndex.categories

**property CategoricalIndex.categories**

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to `categories` is an inplace operation!

**Raises**

`ValueError` If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

**See also:**

- `rename_categories` Rename categories.
- `reorder_categories` Reorder categories.
- `add_categories` Add new categories.
- `remove_categories` Remove the specified categories.
- `remove_unused_categories` Remove categories which are not used.
- `set_categories` Set the categories to the specified ones.

pandas.CategoricalIndex.ordered

**property CategoricalIndex.ordered**

Whether the categories have an ordered relationship.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rename_categories</code></td>
<td>Rename categories</td>
</tr>
<tr>
<td><code>reorder_categories</code></td>
<td>Reorder categories as specified in <code>new_categories</code></td>
</tr>
<tr>
<td><code>add_categories</code></td>
<td>Add new categories</td>
</tr>
<tr>
<td><code>remove_categories</code></td>
<td>Remove the specified categories</td>
</tr>
<tr>
<td><code>remove_unused_categories</code></td>
<td>Remove categories which are not used.</td>
</tr>
<tr>
<td><code>set_categories</code></td>
<td>Set the categories to the specified <code>new_categories</code></td>
</tr>
<tr>
<td><code>as_ordered</code></td>
<td>Set the Categorical to be ordered</td>
</tr>
<tr>
<td><code>as_unordered</code></td>
<td>Set the Categorical to be unordered</td>
</tr>
<tr>
<td><code>map</code></td>
<td>Map values using input an input mapping or function</td>
</tr>
</tbody>
</table>
pandas.CategoricalIndex.rename_categories

CategoricalIndex.rename_categories(*args, **kwargs)

Rename categories.

Parameters

new_categories [list-like, dict-like or callable] New categories which will replace old categories.

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.
- callable: a callable that is called on all items in the old categories and whose return values comprise the new categories.

inplace [bool, default False] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Deprecated since version 1.3.0.

Returns

cat [Categorical or None] Categorical with removed categories or None if inplace=True.

Raises

ValueError If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

See also:

reorder_categories Reorder categories.
add_categories Add new categories.
remove_categories Remove the specified categories.
remove_unused_categories Remove categories which are not used.
set_categories Set the categories to the specified ones.

Examples

```python
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through.

```python
>>> c.rename_categories({'a': 'A', 'c': 'C'})
['A', 'A', 'b']
Categories (2, object): ['A', 'b']
```
You may also provide a callable to create the new categories

```python
>>> c.rename_categories(lambda x: x.upper())
['A', 'A', 'B']
Categories (2, object): ['A', 'B']
```

### pandas.CategoricalIndex.reorder_categories

**CategoricalIndex.reorder_categories(*args, **kwargs)**

Reorder categories as specified in `new_categories`

- `new_categories` need to include all old categories and no new category items.

**Parameters**

- `new_categories` [Index-like] The categories in new order.
- `ordered` [bool, optional] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.
- `inplace` [bool, default False] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories. Depreciated since version 1.3.0.

**Returns**

- `cat` [Categorical or None] Categorical with removed categories or None if inplace=True.

**Raises**

- `ValueError` If the new categories do not contain all old category items or any new ones

**See also:**

- `rename_categories` Rename categories.
- `add_categories` Add new categories.
- `remove_categories` Remove the specified categories.
- `remove_unused_categories` Remove categories which are not used.
- `set_categories` Set the categories to the specified ones.

### pandas.CategoricalIndex.add_categories

**CategoricalIndex.add_categories(*args, **kwargs)**

Add new categories.

- `new_categories` will be included at the last/highest place in the categories and will be unused directly after this call.

**Parameters**

- `new_categories` [category or list-like of category] The new categories to be included.
pandas: powerful Python data analysis toolkit, Release 1.4.2

inplace [bool, default False] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical or None] Categorical with new categories added or None if inplace=True.

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories Rename categories.
reorder_categories Reorder categories.
remove_categories Remove the specified categories.
remove_unused_categories Remove categories which are not used.
set_categories Set the categories to the specified ones.

Examples

```python
>>> c = pd.Categorical(['c', 'b', 'c'])
```
```python
>>> c
['c', 'b', 'c']
Categories (2, object): ['b', 'c']
```
```python
>>> c.add_categories(['d', 'a'])
['c', 'b', 'c']
Categories (4, object): ['b', 'c', 'd', 'a']
```

pandas.CategoricalIndex.remove_categories

CategoricalIndex.remove_categories(*args, **kwargs)
Remove the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN.

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [bool, default False] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical or None] Categorical with removed categories or None if inplace=True.

Raises
**ValueError** If the removals are not contained in the categories

See also:

rename_categories Rename categories.

reorder_categories Reorder categories.

add_categories Add new categories.

remove_unused_categories Remove categories which are not used.

set_categories Set the categories to the specified ones.

**Examples**

```python
>>> c = pd.Categorical(['a', 'c', 'b', 'c', 'd'])
>>> c
['a', 'c', 'b', 'c', 'd']
Categories (4, object): ['a', 'b', 'c', 'd']

>>> c.remove_categories(['d', 'a'])
[NaN, 'c', 'b', 'c', NaN]
Categories (2, object): ['b', 'c']
```

**pandas.CategoricalIndex.remove_unused_categories**

CategoricalIndex.remove_unused_categories(*args, **kwargs)

Remove categories which are not used.

**Parameters**

inplace [bool, default False] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Deprecated since version 1.2.0.

**Returns**

cat [Categorical or None] Categorical with unused categories dropped or None if inplace=True.

See also:

rename_categories Rename categories.

reorder_categories Reorder categories.

add_categories Add new categories.

remove_categories Remove the specified categories.

set_categories Set the categories to the specified ones.
Examples

```python
>>> c = pd.Categorical(['a', 'c', 'b', 'c', 'd'])
```

```
Categories (4, object): ['a', 'b', 'c', 'd']
```

```python
>>> c[2] = 'a'
```

```
Categories (4, object): ['a', 'b', 'a', 'c', 'd']
```

```python
>>> c[4] = 'c'
```

```
Categories (4, object): ['a', 'b', 'a', 'c', 'c']
```

```python
>>> c.remove_unused_categories()
```

```
Categories (2, object): ['a', 'c']
```

```
pandas.CategoricalIndex.set_categories
```

CategoricalIndex.set_categories(*args, **kwargs)

Set the categories to the specified new_categories.

New_categories can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If rename==True, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes, which does not considers a S1 string equal to a single char python string.

**Parameters**

- **new_categories** [Index-like] The categories in new order.

- **ordered** [bool, default False] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

- **rename** [bool, default False] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

- **inplace** [bool, default False] Whether or not to reorder the categories in-place or return a copy of this categorical with reordered categories.

    Deprecated since version 1.3.0.

**Returns**

Categorical with reordered categories or None if inplace.

**Raises**

- **ValueError** If new_categories does not validate as categories

**See also:**
rename_categories  Rename categories.
reorder_categories  Reorder categories.
add_categories  Add new categories.
remove_categories  Remove the specified categories.
remove_unused_categories  Remove categories which are not used.

pandas.CategoricalIndex.as_ordered

CategoricalIndex.as_ordered(*args, **kwargs)
Set the Categorical to be ordered.

Parameters

inplace  [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to True.

Returns

Categorical or None  Ordered Categorical or None if inplace=True.

pandas.CategoricalIndex.as_unordered

CategoricalIndex.as_unordered(*args, **kwargs)
Set the Categorical to be unordered.

Parameters

inplace  [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to False.

Returns

Categorical or None  Unordered Categorical or None if inplace=True.

pandas.CategoricalIndex.map

CategoricalIndex.map(mapper)
Map values using input an input mapping or function.
Maps the values (their categories, not the codes) of the index to new categories. If the mapping correspondence is one-to-one the result is a CategoricalIndex which has the same order property as the original, otherwise an Index is returned.

If a dict or Series is used any unmapped category is mapped to NaN. Note that if this happens an Index will be returned.

Parameters

mapper  [function, dict, or Series] Mapping correspondence.

Returns

pandas.CategoricalIndex or pandas.Index  Mapped index.

See also:
**Index.map**  Apply a mapping correspondence on an Index.

**Series.map**  Apply a mapping correspondence on a Series.

**Series.apply**  Apply more complex functions on a Series.

**Examples**

```python
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'])
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'], ordered=False, dtype='category')

>>> idx.map(lambda x: x.upper())
CategoricalIndex(['A', 'B', 'C'], categories=['A', 'B', 'C'], ordered=False, dtype='category')

>>> idx.map({'a': 'first', 'b': 'second', 'c': 'third'})
CategoricalIndex(['first', 'second', 'third'], categories=['first', 'second', 'third'], ordered=False, dtype='category')
```

If the mapping is one-to-one the ordering of the categories is preserved:

```python
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'], ordered=True)
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'], ordered=True, dtype='category')

>>> idx.map({'a': 3, 'b': 2, 'c': 1})
CategoricalIndex([3, 2, 1], categories=[3, 2, 1], ordered=True, dtype='category')
```

If the mapping is not one-to-one an Index is returned:

```python
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'first'})
Index(['first', 'second', 'first'], dtype='object')
```

If a dict is used, all unmapped categories are mapped to NaN and the result is an Index:

```python
>>> idx.map({'a': 'first', 'b': 'second'})
Index(['first', 'second', nan], dtype='object')
```
**Categorical components**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CategoricalIndex.codes</code></td>
<td>The category codes of this categorical.</td>
</tr>
<tr>
<td><code>CategoricalIndex.categories</code></td>
<td>The categories of this categorical.</td>
</tr>
<tr>
<td><code>CategoricalIndex.ordered</code></td>
<td>Whether the categories have an ordered relationship.</td>
</tr>
<tr>
<td><code>CategoricalIndex.rename_categories</code></td>
<td>Rename categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.reorder_categories</code></td>
<td>Reorder categories as specified in new_categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.add_categories</code></td>
<td>Add new categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.remove_categories</code></td>
<td>Remove the specified categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.remove_unused_categories</code></td>
<td>Remove categories which are not used.</td>
</tr>
<tr>
<td><code>CategoricalIndex.set_categories</code></td>
<td>Set the categories to the specified new_categories.</td>
</tr>
<tr>
<td><code>CategoricalIndex.as_ordered</code></td>
<td>Set the Categorical to be ordered.</td>
</tr>
<tr>
<td><code>CategoricalIndex.as_unordered</code></td>
<td>Set the Categorical to be unordered.</td>
</tr>
</tbody>
</table>

**Modifying and computations**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CategoricalIndex.map</code></td>
<td>Map values using input an input mapping or function.</td>
</tr>
<tr>
<td><code>CategoricalIndex.equals</code></td>
<td>Determine if two CategoricalIndex objects contain the same elements.</td>
</tr>
</tbody>
</table>

**pandas.CategoricalIndex.equals**

`CategoricalIndex.equals(other)`

Determine if two CategoricalIndex objects contain the same elements.

**Returns**

- `bool` If two CategoricalIndex objects have equal elements True, otherwise False.

### 3.6.4 `IntervalIndex`

**IntervalIndex**(data[, closed, dtype, copy, ...])

Immutable index of intervals that are closed on the same side.

**pandas.IntervalIndex**

```python
class pandas.IntervalIndex(data, closed=None, dtype=None, copy=False, name=None, verify_integrity=True)
```

Immutable index of intervals that are closed on the same side.

New in version 0.20.0.

**Parameters**

- `data` [array-like (1-dimensional)] Array-like containing Interval objects from which to build the IntervalIndex.
closed  [[‘left’, ‘right’, ‘both’, ‘neither’], default ‘right’] Whether the intervals are closed on the left-side, right-side, both or neither.
dtype  [dtype or None, default None] If None, dtype will be inferred.
copy  [bool, default False] Copy the input data.
name  [object, optional] Name to be stored in the index.

verify_integrity  [bool, default True] Verify that the IntervalIndex is valid.

See also:

Index  The base pandas Index type.
Interval  A bounded slice-like interval; the elements of an IntervalIndex.
interval_range  Function to create a fixed frequency IntervalIndex.
cut  Bin values into discrete Intervals.
qcut  Bin values into equal-sized Intervals based on rank or sample quantiles.

Notes

See the user guide for more.

Examples

A new IntervalIndex is typically constructed using interval_range():

```python
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
dtype='interval[int64, right]')
```

It may also be constructed using one of the constructor methods: IntervalIndex.from_arrays(), IntervalIndex.from_breaks(), and IntervalIndex.from_tuples().

See further examples in the doc strings of interval_range and the mentioned constructor methods.

Attributes

<table>
<thead>
<tr>
<th>closed</th>
<th>Whether the intervals are closed on the left-side, right-side, both or neither.</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_empty</td>
<td>Indic平es if an interval is empty, meaning it contains no points.</td>
</tr>
<tr>
<td>is_non_overlapping_monotonic</td>
<td>Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonically increasing or monotonically decreasing, else False.</td>
</tr>
<tr>
<td>is_overlapping</td>
<td>Return True if the IntervalIndex has overlapping intervals, else False.</td>
</tr>
<tr>
<td>values</td>
<td>Return an array representing the data in the Index.</td>
</tr>
</tbody>
</table>
pandas.IntervalIndex.closed

IntervalIndex.closed
Whether the intervals are closed on the left-side, right-side, both or neither.

pandas.IntervalIndex.is_empty

property IntervalIndex.is_empty
Indicates if an interval is empty, meaning it contains no points.
New in version 0.25.0.

Returns

bool or ndarray A boolean indicating if a scalar Interval is empty, or a boolean ndarray positionally indicating if an Interval in an IntervalArray or IntervalIndex is empty.

Examples

An Interval that contains points is not empty:

```python
>>> pd.Interval(0, 1, closed='right').is_empty
False
```

An Interval that does not contain any points is empty:

```python
>>> pd.Interval(0, 0, closed='right').is_empty
True
>>> pd.Interval(0, 0, closed='left').is_empty
True
>>> pd.Interval(0, 0, closed='neither').is_empty
True
```

An Interval that contains a single point is not empty:

```python
>>> pd.Interval(0, 0, closed='both').is_empty
False
```

An IntervalArray or IntervalIndex returns a boolean ndarray positionally indicating if an Interval is empty:

```python
>>> ivs = [pd.Interval(0, 0, closed='neither'),
          pd.Interval(1, 2, closed='neither')]
>>> pd.arrays.IntervalArray(ivs).is_empty
array([ True, False])
```

Missing values are not considered empty:

```python
>>> ivs = [pd.Interval(0, 0, closed='neither'), np.nan]
>>> pd.IntervalIndex(ivs).is_empty
array([ True, False])
```
pandas.IntervalIndex.is_non_overlapping_monotonic

IntervalIndex.is_non_overlapping_monotonic
Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False.

pandas.IntervalIndex.is_overlapping

property IntervalIndex.is_overlapping
Return True if the IntervalIndex has overlapping intervals, else False.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

Returns
bool Boolean indicating if the IntervalIndex has overlapping intervals.

See also:
Interval.overlaps Check whether two Interval objects overlap.
IntervalIndex.overlaps Check an IntervalIndex elementwise for overlaps.

Examples

```python
>>> index = pd.IntervalIndex.from_tuples([(0, 2), (1, 3), (4, 5)])
>>> index
IntervalIndex([(0, 2), (1, 3), (4, 5)],
dtype='interval[int64, right]')
>>> index.is_overlapping
True
```

Intervals that share closed endpoints overlap:

```python
>>> index = pd.interval_range(0, 3, closed='both')
>>> index
IntervalIndex([(0, 1), [1, 2], [2, 3]],
dtype='interval[int64, both]')
>>> index.is_overlapping
True
```

Intervals that only have an open endpoint in common do not overlap:

```python
>>> index = pd.interval_range(0, 3, closed='left')
>>> index
IntervalIndex([(0, 1), [1, 2], [2, 3]],
dtype='interval[int64, left]')
>>> index.is_overlapping
False
```
pandas.IntervalIndex.values

property IntervalIndex.values
Return an array representing the data in the Index.

Warning: We recommend using Index.array or Index.to_numpy(), depending on whether you need a reference to the underlying data or a NumPy array.

Returns
array: numpy.ndarray or ExtensionArray

See also:
Index.array Reference to the underlying data.
Index.to_numpy A NumPy array representing the underlying data.

left  
right  
mid  
length

Methods

from_arrays(left, right[, closed, name, ...]) Construct from two arrays defining the left and right bounds.

from_tuples(data[, closed, name, copy, dtype]) Construct an IntervalIndex from an array-like of tuples.

from_breaks(breaks[, closed, name, copy, dtype]) Construct an IntervalIndex from an array of splits.

contains(*args, **kwargs) Check elementwise if the Intervals contain the value.

overlaps(*args, **kwargs) Check elementwise if an Interval overlaps the values in the IntervalArray.

set_closed(*args, **kwargs) Return an IntervalArray identical to the current one, but closed on the specified side.

to_tuples(*args, **kwargs) Return an ndarray of tuples of the form (left, right).

pandas.IntervalIndex.from_arrays

classmethod IntervalIndex.from_arrays(left, right, closed='right', name=None, copy=False, dtype=None)
Construct from two arrays defining the left and right bounds.

Parameters
left [array-like (1-dimensional)] Left bounds for each interval.
right [array-like (1-dimensional)] Right bounds for each interval.
closed [{‘left’, ‘right’, ‘both’, ‘neither’}, default ‘right’] Whether the intervals are closed on the left-side, right-side, both or neither.
copy [bool, default False] Copy the data.
dtype [dtype, optional] If None, dtype will be inferred.

Returns
IntervalIndex

Raises

ValueError When a value is missing in only one of left or right. When a value in left is greater than the corresponding value in right.

See also:

interval_range Function to create a fixed frequency IntervalIndex.
IntervalIndex.from_breaks Construct an IntervalIndex from an array of splits.
IntervalIndex.from_tuples Construct an IntervalIndex from an array-like of tuples.

Notes

Each element of left must be less than or equal to the right element at the same position. If an element is missing, it must be missing in both left and right. A TypeError is raised when using an unsupported type for left or right. At the moment, ‘category’, ‘object’, and ‘string’ subtypes are not supported.

Examples

```python
>>> pd.IntervalIndex.from_arrays([0, 1, 2], [1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
             dtype='interval[int64, right]')
```

pandas.IntervalIndex.from_tuples

classmethod IntervalIndex.from_tuples(data, closed='right', name=None, copy=False, dtype=None)

Construct an IntervalIndex from an array-like of tuples.

Parameters

data [array-like (1-dimensional)] Array of tuples.
closed ['left', 'right', 'both', 'neither'], default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.
copy [bool, default False] By-default copy the data, this is compat only and ignored.
dtype [dtype or None, default None] If None, dtype will be inferred.

Returns

IntervalIndex

See also:

interval_range Function to create a fixed frequency IntervalIndex.
IntervalIndex.from_breaks Construct an IntervalIndex from an array of splits.
IntervalIndex.from_arrays Construct an IntervalIndex from a left and right array.
Examples

```python
>>> pd.IntervalIndex.from_tuples(((0, 1), (1, 2)))
IntervalIndex([(0, 1], (1, 2]],
   dtype='interval[int64, right]')
```

pandas.IntervalIndex.from_breaks

**classmethod IntervalIndex.from_breaks**(breaks, closed='right', name=None, copy=False, dtype=None)

Construct an IntervalIndex from an array of splits.

**Parameters**

- **breaks**  [array-like (1-dimensional)] Left and right bounds for each interval.
- **closed**  [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.
- **copy**  [bool, default False] Copy the data.
- **dtype**  [dtype or None, default None] If None, dtype will be inferred.

**Returns**

IntervalIndex

See also:

- `interval_range` Function to create a fixed frequency IntervalIndex.
- `IntervalIndex.from_arrays` Construct from a left and right array.
- `IntervalIndex.from_tuples` Construct from a sequence of tuples.

Examples

```python
>>> pd.IntervalIndex.from_breaks([0, 1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
   dtype='interval[int64, right]')
```

pandas.IntervalIndex.contains

**IntervalIndex.contains**(other, **kwargs)

Check elementwise if the Intervals contain the value.

Return a boolean mask whether the value is contained in the Intervals of the IntervalArray.

New in version 0.25.0.

**Parameters**

- **other**  [scalar] The value to check whether it is contained in the Intervals.

**Returns**

boolean array
See also:

**Interval.contains** Check whether Interval object contains value.

**IntervalArray.overlaps** Check if an Interval overlaps the values in the IntervalArray.

**Examples**

```python
>>> intervals = pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 3), (2, 4)])
>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]
Length: 3, dtype: interval[int64, right]

>>> intervals.contains(0.5)
array([ True, False, False])
```

**pandas.IntervalIndex.overlaps**

**IntervalIndex.overlaps**(*args, **kwargs)

Check elementwise if an Interval overlaps the values in the IntervalArray.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

**Parameters**

other [IntervalArray] Interval to check against for an overlap.

**Returns**

ndarray Boolean array positionally indicating where an overlap occurs.

See also:

**Interval.overlaps** Check whether two Interval objects overlap.

**Examples**

```python
>>> data = [(0, 1), (1, 3), (2, 4)]
>>> intervals = pd.arrays.IntervalArray.from_tuples(data)
>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]
Length: 3, dtype: interval[int64, right]

>>> intervals.overlaps(pd.Interval(0.5, 1.5))
array([ True, True, False])

Intervals that share closed endpoints overlap:

>>> intervals.overlaps(pd.Interval(1, 3, closed='left'))
array([ True, False, True])
```
Intervals that only have an open endpoint in common do not overlap:

```python
>>> intervals.overlaps(pd.Interval(1, 2, closed='right'))
array([False,  True, False])
```

**pandas.IntervalIndex.set_closed**

`IntervalIndex.set_closed(*args, **kwargs)`

Return an IntervalArray identical to the current one, but closed on the specified side.

**Parameters**

- `closed` [{‘left’, ‘right’, ‘both’, ‘neither’}] Whether the intervals are closed on the left-side, right-side, both or neither.

**Returns**

- `new_index` [IntervalArray]

**Examples**

```python
>>> index = pd.arrays.IntervalArray.from_breaks(range(4))
>>> index
<IntervalArray>
[(0, 1], (1, 2], (2, 3])
Length: 3, dtype: interval[int64, right]
>>> index.set_closed('both')
<IntervalArray>
[[0, 1], [1, 2], [2, 3]]
Length: 3, dtype: interval[int64, both]
```

**pandas.IntervalIndex.to_tuples**

`IntervalIndex.to_tuples(*args, **kwargs)`

Return an ndarray of tuples of the form (left, right).

**Parameters**

- `na_tuple` [bool, default True] Returns NA as a tuple if True, (nan, nan), or just as the NA value itself if False, nan.

**Returns**

- `tuples`: ndarray
**IntervalIndex components**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IntervalIndex.from_arrays(left, right[, ...])</code></td>
<td>Construct from two arrays defining the left and right bounds.</td>
</tr>
<tr>
<td><code>IntervalIndex.from_tuples(data[, closed, ...])</code></td>
<td>Construct an IntervalIndex from an array-like of tuples.</td>
</tr>
<tr>
<td><code>IntervalIndex.from_breaks(breaks[, closed, ...])</code></td>
<td>Construct an IntervalIndex from an array of splits.</td>
</tr>
</tbody>
</table>

- **`IntervalIndex.left`**

- **`IntervalIndex.right`**

- **`IntervalIndex.mid`**

- **`IntervalIndex.closed`** Whether the intervals are closed on the left-side, right-side, both or neither.

- **`IntervalIndex.length`**

- **`IntervalIndex.values`** Return an array representing the data in the Index.

- **`IntervalIndex.is_empty`** Indicates if an interval is empty, meaning it contains no points.

- **`IntervalIndex.is_non_overlapping_monotonic`** Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False.

- **`IntervalIndex.is_overlapping`** Return True if the IntervalIndex has overlapping intervals, else False.

- **`IntervalIndex.get_loc(key[, method, tolerance])`** Get integer location, slice or boolean mask for requested label.

- **`IntervalIndex.get_indexer(target[, method, ...])`** Compute indexer and mask for new index given the current index.

- **`IntervalIndex.set_closed(*args, **kwargs)`** Return an IntervalArray identical to the current one, but closed on the specified side.

- **`IntervalIndex.contains(*args, **kwargs)`** Check elementwise if the Intervals contain the value.

- **`IntervalIndex.overlaps(*args, **kwargs)`** Check elementwise if an Interval overlaps the values in the IntervalArray.

- **`IntervalIndex.to_tuples(*args, **kwargs)`** Return an ndarray of tuples of the form (left, right).
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.IntervalIndex.right**

IntervalIndex.right

**pandas.IntervalIndex.mid**

IntervalIndex.mid

**pandas.IntervalIndex.length**

property IntervalIndex.length

**pandas.IntervalIndex.get_loc**

IntervalIndex.get_loc(key, method=None, tolerance=None)

Get integer location, slice or boolean mask for requested label.

**Parameters**

key [label]

method [{None}, optional]

- default: matches where the label is within an interval only.

**Returns**

int if unique index, slice if monotonic index, else mask

**Examples**

```python
>>> i1, i2 = pd.Interval(0, 1), pd.Interval(1, 2)
>>> index = pd.IntervalIndex([i1, i2])
>>> index.get_loc(1)
0
```

You can also supply a point inside an interval.

```python
>>> index.get_loc(1.5)
1
```

If a label is in several intervals, you get the locations of all the relevant intervals.

```python
>>> i3 = pd.Interval(0, 2)
>>> overlapping_index = pd.IntervalIndex([i1, i2, i3])
>>> overlapping_index.get_loc(0.5)
array([ True, False,  True])
```

Only exact matches will be returned if an interval is provided.

```python
>>> index.get_loc(pd.Interval(0, 1))
0
```
**pandas.IntervalIndex.get_indexer**

IntervalIndex.get_indexer(target, method=None, limit=None, tolerance=None)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

**Parameters**

- **target** [Index]
- **method** [{None, 'pad'/’ffill', 'backfill'/’bfill', 'nearest'}, optional]
  - default: exact matches only.
  - pad / ffill: find the PREVIOUS index value if no exact match.
  - backfill / bfill: use NEXT index value if no exact match
  - nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.
- **limit** [int, optional] Maximum number of consecutive labels in target to match for inexact matches.
- **tolerance** [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation abs(index[indexer] - target) <= tolerance.
  
  Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

**Returns**

- **indexer** [np.ndarray[np.intp]] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

**Notes**

Returns -1 for unmatched values, for further explanation see the example below.

**Examples**

```python
>>> index = pd.Index(["c", "a", "b")
>>> index.get_indexer(["a", "b", "x")
array([ 1, 2, -1])
```

Notice that the return value is an array of locations in index and x is marked by -1, as it is not in index.
3.6.5 MultiIndex

```
MultiIndex([levels, codes, sortorder, ...])  A multi-level, or hierarchical, index object for pandas objects.
```

**pandas.MultiIndex**

class pandas.MultiIndex(levels=None, codes=None, sortorder=None, names=None, dtype=None, copy=False, name=None, verify_integrity=True)

A multi-level, or hierarchical, index object for pandas objects.

**Parameters**

- **levels** [sequence of arrays] The unique labels for each level.
- **codes** [sequence of arrays] Integers for each level designating which label at each location.
- **sortorder** [optional int] Level of sortedness (must be lexicographically sorted by that level).
- **names** [optional sequence of objects] Names for each of the index levels. (name is accepted for compat).
- **copy** [bool, default False] Copy the meta-data.
- **verify_integrity** [bool, default True] Check that the levels/codes are consistent and valid.

**See also:**
- MultiIndex.from_arrays Convert list of arrays to MultiIndex.
- MultiIndex.from_product Create a MultiIndex from the cartesian product of iterables.
- MultiIndex.from_tuples Convert list of tuples to a MultiIndex.
- MultiIndex.from_frame Make a MultiIndex from a DataFrame.
- Index The base pandas Index type.

**Notes**

See the user guide for more.

**Examples**

A new MultiIndex is typically constructed using one of the helper methods MultiIndex.from_arrays(), MultiIndex.from_product() and MultiIndex.from_tuples(). For example (using .from_arrays):

```python
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex([(1, 'red'),
           (1, 'blue'),
           (2, 'red'),
           (2, 'blue')],
          names=['number', 'color'])
```

See further examples for how to construct a MultiIndex in the doc strings of the mentioned helper methods.
Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>Names of levels in MultiIndex.</td>
</tr>
<tr>
<td>nlevels</td>
<td>Integer number of levels in this MultiIndex.</td>
</tr>
<tr>
<td>levshape</td>
<td>A tuple with the length of each level.</td>
</tr>
</tbody>
</table>

**pandas.MultiIndex.names**

**property** `MultiIndex.names`

Names of levels in MultiIndex.

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays(... [[1, 2], [3, 4], [5, 6]], names=['x', 'y', 'z'])
>>> mi
MultiIndex([(1, 3, 5),
            (2, 4, 6)],
           names=['x', 'y', 'z'])
>>> mi.names
FrozenList(['x', 'y', 'z'])
```

**pandas.MultiIndex.nlevels**

**property** `MultiIndex.nlevels`

Integer number of levels in this MultiIndex.

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays([['a'], ['b'], ['c']])
>>> mi
MultiIndex([['a', 'b', 'c']])
>>> mi.nlevels
3
```

**pandas.MultiIndex.levshape**

**property** `MultiIndex.levshape`

A tuple with the length of each level.
Examples

```python
>>> mi = pd.MultiIndex.from_arrays([['a'], ['b'], ['c']])
>>> mi
MultiIndex([('a', 'b', 'c')],
            )
>>> mi.levshape
(1, 1, 1)
```

<table>
<thead>
<tr>
<th>levels</th>
<th>codes</th>
</tr>
</thead>
</table>

Methods

- `from_arrays(arrays[, sortorder, names])` Convert arrays to MultiIndex.
- `from_tuples(tuples[, sortorder, names])` Convert list of tuples to MultiIndex.
- `from_product(iterables[, sortorder, names])` Make a MultiIndex from the cartesian product of multiple iterables.
- `from_frame(df[, sortorder, names])` Make a MultiIndex from a DataFrame.
- `set_levels(levels[, level, inplace, ...])` Set new levels on MultiIndex.
- `set_codes(codes[, level, inplace, ...])` Set new codes on MultiIndex.
- `to_frame([index, name])` Create a DataFrame with the levels of the MultiIndex as columns.
- `to_flat_index()` Convert a MultiIndex to an Index of Tuples containing the level values.
- `sortlevel([level, ascending, sort_remaining])` Sort MultiIndex at the requested level.
- `droplevel([level])` Return index with requested level(s) removed.
- `swaplevel([i, j])` Swap level `i` with level `j`.
- `reorder_levels(order)` Rearrange levels using input order.
- `remove_unused_levels()` Create new MultiIndex from current that removes unused levels.
- `get_locs(seq)` Get location for a sequence of labels.

`pandas.MultiIndex.from_arrays`

**classmethod** `MultiIndex.from_arrays(arrays, sortorder=None, names=NoDefault.no_default)` Convert arrays to MultiIndex.

**Parameters**

- `arrays` [list / sequence of array-likes] Each array-like gives one level’s value for each data point. len(arrays) is the number of levels.
- `sortorder` [int or None] Level of sortedness (must be lexicographically sorted by that level).
- `names` [list / sequence of str, optional] Names for the levels in the index.

**Returns**

MultiIndex

See also:
pandas: powerful Python data analysis toolkit, Release 1.4.2

**MultiIndex.from_tuples**  Convert list of tuples to MultiIndex.

**MultiIndex.from_product**  Make a MultiIndex from cartesian product of iterables.

**MultiIndex.from_frame**  Make a MultiIndex from a DataFrame.

**Examples**

```python
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

pandas.MultiIndex.from_tuples

classmethod MultiIndex.from_tuples(tuples, sortorder=None, names=None)

Convert list of tuples to MultiIndex.

**Parameters**

- **tuples**  [list / sequence of tuple-likes] Each tuple is the index of one row/column.
- **sortorder**  [int or None] Level of sortedness (must be lexicographically sorted by that level).
- **names**  [list / sequence of str, optional] Names for the levels in the index.

**Returns**

MultiIndex

**See also:**

- **MultiIndex.from_arrays**  Convert list of arrays to MultiIndex.
- **MultiIndex.from_product**  Make a MultiIndex from cartesian product of iterables.
- **MultiIndex.from_frame**  Make a MultiIndex from a DataFrame.

**Examples**

```python
>>> tuples = [(1, 'red'), (1, 'blue'),
            (2, 'red'), (2, 'blue')]
>>> pd.MultiIndex.from_tuples(tuples, names=('number', 'color'))
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
           names=['number', 'color'])
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.MultiIndex.from_product

**classmethod** MultiIndex.from_product(iterables, sortorder=None, names=NoDefault.no_default)

Make a MultiIndex from the cartesian product of multiple iterables.

**Parameters**

iterables  [list / sequence of iterables] Each iterable has unique labels for each level of the index.

sortorder  [int or None] Level of sortedness (must be lexicographically sorted by that level).

names  [list / sequence of str, optional] Names for the levels in the index.

Changed in version 1.0.0: If not explicitly provided, names will be inferred from the elements of iterables if an element has a name attribute

**Returns**

MultiIndex

See also:

MultiIndex.from_arrays  Convert list of arrays to MultiIndex.

MultiIndex.from_tuples  Convert list of tuples to MultiIndex.

MultiIndex.from_frame  Make a MultiIndex from a DataFrame.

**Examples**

```python
>>> numbers = [0, 1, 2]
>>> colors = ['green', 'purple']
>>> pd.MultiIndex.from_product([numbers, colors], names=['number', 'color'])
MultiIndex([(0, 'green'),
           (0, 'purple'),
           (1, 'green'),
           (1, 'purple'),
           (2, 'green'),
           (2, 'purple')],
          names=['number', 'color'])
```
names  [list-like, optional] If no names are provided, use the column names, or tuple of column names if the columns is a MultiIndex. If a sequence, overwrite names with the given sequence.

Returns

MultiIndex  The MultiIndex representation of the given DataFrame.

See also:

MultiIndex.from_arrays  Convert list of arrays to MultiIndex.
MultiIndex.from_tuples  Convert list of tuples to MultiIndex.
MultiIndex.from_product  Make a MultiIndex from cartesian product of iterables.

Examples

```python
>>> df = pd.DataFrame([[['HI', 'Temp'], ['HI', 'Precip']],
...                     [['NJ', 'Temp'], ['NJ', 'Precip']],
...                     columns=['a', 'b'])
```

```python
>>> df
    a   b
0   HI Temp
1   HI Precip
2   NJ Temp
3   NJ Precip
```

```python
>>> pd.MultiIndex.from_frame(df)
MultiIndex([('HI', 'Temp'), ('HI', 'Precip'), ('NJ', 'Temp'), ('NJ', 'Precip')], names=['a', 'b'])
```

Using explicit names, instead of the column names

```python
>>> pd.MultiIndex.from_frame(df, names=['state', 'observation'])
MultiIndex([('HI', 'Temp'), ('HI', 'Precip'), ('NJ', 'Temp'), ('NJ', 'Precip')], names=['state', 'observation'])
```

pandas.MultiIndex.set_levels

MultiIndex.set_levels(levels, level=None, inplace=None, verify_integrity=True)

Set new levels on MultiIndex. Defaults to returning new index.

Parameters

levels  [sequence or list of sequence] New level(s) to apply.

level  [int, level name, or sequence of int/level names (default None)] Level(s) to set (None for all levels).
inplace  [bool] If True, mutates in place.

Depreciated since version 1.2.0.

verify_integrity  [bool, default True] If True, checks that levels and codes are compatible.

Returns

new index (of same type and class...etc) or None  The same type as the caller or None if inplace=True.

Examples

```python
>>> idx = pd.MultiIndex.from_tuples(
...    [(1, "one"),
...     (1, "two"),
...     (2, "one"),
...     (2, "two"),
...     (3, "one"),
...     (3, "two")
... ],
...    names=['foo', 'bar']
... )
>>> idx
MultiIndex([(1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two'),
            (3, 'one'),
            (3, 'two')],
           names=['foo', 'bar'])
```

```python
>>> idx.set_levels([['a', 'b', 'c'], [1, 2]])
MultiIndex([['a', 1],
           ['b', 1],
           ['b', 2],
           ['c', 1],
           ['c', 2]],
          names=['foo', 'bar'])
```

```python
>>> idx.set_levels(['a', 'b', 'c'], level=0)
MultiIndex([['a', 'one'],
            ['a', 'two'],
            ['b', 'one'],
            ['b', 'two'],
            ['c', 'one'],
            ['c', 'two']],
           names=['foo', 'bar'])
```

```python
>>> idx.set_levels(['a', 'b'], level='bar')
MultiIndex([(1, 'a'),
            (1, 'b'),
            (2, 'a')],
           names=['foo', 'bar'])
```

(continues on next page)
If any of the levels passed to `set_levels()` exceeds the existing length, all of the values from that argument will be stored in the MultiIndex levels, though the values will be truncated in the MultiIndex output.

```
>>> idx.set_levels([['a', 'b', 'c'], [1, 2, 3, 4]], level=[0, 1])
MultiIndex([['a', 'b', 'c'], [1, 2, 3, 4]], level=[0, 1])

>>> idx.set_levels([['a', 'b', 'c'], [1, 2, 3, 4]], level=[0, 1]).levels
FrozenList([['a', 'b', 'c'], [1, 2, 3, 4]])
```

### pandas.MultiIndex.set_codes

`MultiIndex.set_codes(codes, level=None, inplace=None, verify_integrity=True)`

Set new codes on MultiIndex. Defaults to returning new index.

**Parameters**

- **codes** [sequence or list of sequence] New codes to apply.
- **level** [int, level name, or sequence of int/level names (default None)] Level(s) to set (None for all levels).
- **inplace** [bool] If True, mutates in place.
  
  Deprecated since version 1.2.0.

- **verify_integrity** [bool, default True] If True, checks that levels and codes are compatible.

**Returns**

- **new index (of same type and class...etc) or None** The same type as the caller or None if inplace=True.

### Examples

```
>>> idx = pd.MultiIndex.from_tuples(
...     [(1, 'one'), (1, 'two'), (2, 'one'), (2, 'two')], names=['foo', 'bar']
... )

>>> idx
MultiIndex([(1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['foo', 'bar'])
```
```python
>>> idx.set_codes([[1, 0, 1, 0], [0, 0, 1, 1]])
MultiIndex(((2, 'one'),
            (1, 'one'),
            (2, 'two'),
            (1, 'two')),
           names=['foo', 'bar'])
>>> idx.set_codes([[1, 0, 1, 0], level=0])
MultiIndex(((2, 'one'),
            (1, 'two'),
            (2, 'one'),
            (1, 'two')),
           names=['foo', 'bar'])
>>> idx.set_codes([[0, 0, 1, 1], level='bar'])
MultiIndex(((1, 'one'),
            (1, 'one'),
            (2, 'two'),
            (2, 'two')),
           names=['foo', 'bar'])
>>> idx.set_codes([[1, 0, 1, 0], [0, 0, 1, 1], level=[0, 1]])
MultiIndex(((2, 'one'),
            (1, 'one'),
            (2, 'two'),
            (1, 'two')),
           names=['foo', 'bar'])
```

**pandas.MultiIndex.to_frame**

`MultiIndex.to_frame(index=True, name=NoDefault.no_default)`

Create a DataFrame with the levels of the MultiIndex as columns.

Column ordering is determined by the DataFrame constructor with data as a dict.

**Parameters**

- `index` [bool, default True] Set the index of the returned DataFrame as the original MultiIndex.
- `name` [list / sequence of str, optional] The passed names should substitute index level names.

**Returns**

- DataFrame [a DataFrame containing the original MultiIndex data.]

**See also:**

- `DataFrame` Two-dimensional, size-mutable, potentially heterogeneous tabular data.
Examples

```python
>>> mi = pd.MultiIndex.from_arrays([['a', 'b'], ['c', 'd']])
>>> mi
MultiIndex([('a', 'c'), ('b', 'd')])
```
```
>>> df = mi.to_frame()
>>> df
     0 1
   0 a c
   1 b d
```
```
>>> df = mi.to_frame(index=False)
>>> df
   0 1
   0 a c
   1 b d
```
```
>>> df = mi.to_frame(name=['x', 'y'])
>>> df
   x y
   a c
   b d
```

**pandas.MultiIndex.to_flat_index**

`MultiIndex.to_flat_index()`  
Convert a MultiIndex to an Index of Tuples containing the level values.

**Returns**

- `pd.Index`  Index with the MultiIndex data represented in Tuples.

**See also:**

- `MultiIndex.from_tuples`  Convert flat index back to MultiIndex.

**Notes**

This method will simply return the caller if called by anything other than a MultiIndex.
Examples

```python
>>> index = pd.MultiIndex.from_product(
...    [['foo', 'bar'], ['baz', 'qux']],
...    names=['a', 'b'])
>>> index.to_flat_index()
Index([('foo', 'baz'), ('foo', 'qux'), ('bar', 'baz'), ('bar', 'qux')],
    dtype='object')
```

### pandas.MultiIndex.sortlevel

MultiIndex.sortlevel(level=0, ascending=True, sort_remaining=True)

Sort MultiIndex at the requested level.

The result will respect the original ordering of the associated factor at that level.

**Parameters**

- **level** [list-like, int or str, default 0] If a string is given, must be a name of the level. If list-like must be names or ints of levels.
- **ascending** [bool, default True] False to sort in descending order. Can also be a list to specify a directed ordering.
- **sort_remaining** [sort by the remaining levels after level]

**Returns**

- **sorted_index** [pd.MultiIndex] Resulting index.
- **indexer** [np.ndarray[np.intp]] Indices of output values in original index.

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays([[0, 0], [2, 1]])
>>> mi
MultiIndex([(0, 2),
            (0, 1)],
           names=['a', 'b'])
>>> mi.sortlevel()
(MultiIndex([(0, 1),
              (0, 2)],
             names=['a', 'b'])), array([1, 0])
>>> mi.sortlevel(sort_remaining=False)
(MultiIndex([(0, 2),
              (0, 1)],
             names=['a', 'b'])), array([0, 1])
>>> mi.sortlevel(1)
(MultiIndex([(0, 1),
              (0, 2)],
             names=['a', 'b'])), array([0, 1])
```

(continues on next page)
(continued from previous page)

```python
>>> mi.sortlevel(1, ascending=False)
(MultiIndex([(0, 2),
             (0, 1)],
            ), array([0, 1]))
```

**pandas.MultiIndex.droplevel**

`MultiIndex.droplevel(level=0)`

Return index with requested level(s) removed.

If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

**Parameters**

- `level` [int, str, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

**Returns**

- Index or MultiIndex

**Examples**

```python
>>> mi = pd.MultiIndex.from_arrays(...
... [[1, 2], [3, 4], [5, 6]], names=['x', 'y', 'z'])
>>> mi
MultiIndex([(1, 3, 5),
            (2, 4, 6)],
           names=['x', 'y', 'z'])

>>> mi.droplevel()  
MultiIndex([(3, 5),
            (4, 6)],
           names=['y', 'z'])

>>> mi.droplevel(2)  
MultiIndex([(1, 3),
            (2, 4)],
           names=['x', 'y'])

>>> mi.droplevel('z')  
MultiIndex([(1, 3),
            (2, 4)],
           names=['x', 'y'])

>>> mi.droplevel(['x', 'y'])  
Int64Index([5, 6], dtype='int64', name='z')
```
**pandas.MultiIndex.swaplevel**

`MultiIndex.swaplevel(i=-2, j=-1)`  
Swap level `i` with level `j`.  
Calling this method does not change the ordering of the values.

**Parameters**

- `i` [int, str, default -2] First level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.
- `j` [int, str, default -1] Second level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

**Returns**

`MultiIndex` A new MultiIndex.

**See also:**

- `Series.swaplevel` Swap levels `i` and `j` in a MultiIndex.
- `Dataframe.swaplevel` Swap levels `i` and `j` in a MultiIndex on a particular axis.

**Examples**

```python
>>> mi = pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
                    codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi
MultiIndex([('a', 'bb'), ('a', 'aa'), ('b', 'bb'), ('b', 'aa')], )
>>> mi.swaplevel(0, 1)
MultiIndex([('bb', 'a'), ('aa', 'a'), ('bb', 'b'), ('aa', 'b')], )
```

**pandas.MultiIndex.reorder_levels**

`MultiIndex.reorder_levels(order)`  
Rearrange levels using input order. May not drop or duplicate levels.

**Parameters**

- `order` [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

**Returns**

`MultiIndex`
Examples

```python
>>> mi = pd.MultiIndex.from_arrays([[1, 2], [3, 4]], names=['x', 'y'])
>>> mi
MultiIndex([(1, 3),
            (2, 4)],
           names=['x', 'y'])
```

```python
>>> mi.reorder_levels(order=[1, 0])
MultiIndex([(3, 1),
            (4, 2)],
           names=['y', 'x'])
```

```python
>>> mi.reorder_levels(order=['y', 'x'])
MultiIndex([(3, 1),
            (4, 2)],
           names=['y', 'x'])
```

`pandas.MultiIndex.remove_unused_levels`

`MultiIndex.remove_unused_levels()`

Create new MultiIndex from current that removes unused levels.

Unused level(s) means levels that are not expressed in the labels. The resulting MultiIndex will have the same outward appearance, meaning the same values and ordering. It will also be .equals() to the original.

Returns

`MultiIndex`

Examples

```python
>>> mi = pd.MultiIndex.from_product([range(2), list('ab')])
>>> mi
MultiIndex([(0, 'a'),
            (0, 'b'),
            (1, 'a'),
            (1, 'b')],
           )
```

```python
>>> mi[2:]
MultiIndex([(1, 'a'),
            (1, 'b')],
           )
```

The 0 from the first level is not represented and can be removed

```python
>>> mi2 = mi[2:].remove_unused_levels()
>>> mi2.levels
FrozenList([[1], ['a', 'b']])
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.MultiIndex.get_locs

MultiIndex.get_locs(seq)
Get location for a sequence of labels.

Parameters

seq [label, slice, list, mask or a sequence of such] You should use one of the above for each level. If a level should not be used, set it to slice(None).

Returns

numpy.ndarray NumPy array of integers suitable for passing to iloc.

See also:

MultiIndex.get_loc Get location for a label or a tuple of labels.
MultiIndex.slice_locs Get slice location given start label(s) and end label(s).

Examples

```python
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
```

```python
>>> mi.get_locs('b')
array([1, 2], dtype=int64)
```

```python
>>> mi.get_locs([slice(None), ['e', 'f']])
array([1, 2], dtype=int64)
```

```python
>>> mi.get_locs([[True, False, True], slice('e', 'f')])
array([2], dtype=int64)
```

---

IndexSlice
Create an object to more easily perform multi-index slicing.

pandas.IndexSlice

pandas.IndexSlice = <pandas.core.indexing._IndexSlice object>
Create an object to more easily perform multi-index slicing.

See also:

MultiIndex.remove_unused_levels New MultiIndex with no unused levels.
Notes

See Defined Levels for further info on slicing a MultiIndex.

Examples

```python
>>> midx = pd.MultiIndex.from_product([['A0', 'A1'], ['B0', 'B1', 'B2', 'B3']])
>>> columns = ['foo', 'bar']
>>> dfmi = pd.DataFrame(np.arange(16).reshape((len(midx), len(columns))),
                        index=midx, columns=columns)

Using the default slice command:

```python
>>> dfmi.loc[(slice(None), slice('B0', 'B1')), :]
foo  bar
   A0 B0  0  1
   B1  2  3
   A1 B0  8  9
   B1 10 11
```

Using the IndexSlice class for a more intuitive command:

```python
>>> idx = pd.IndexSlice
>>> dfmi.loc[idx[:, 'B0':'B1'], :]
foo  bar
   A0 B0  0  1
   B1  2  3
   A1 B0  8  9
   B1 10 11
```

MultiIndex constructors

- `MultiIndex.from_arrays(arrays[, sortorder, ...])`: Convert arrays to MultiIndex.
- `MultiIndex.from_tuples(tuples[, sortorder, ...])`: Convert list of tuples to MultiIndex.
- `MultiIndex.from_product(iterables[, ...])`: Make a MultiIndex from the cartesian product of multiple iterables.
- `MultiIndex.from_frame(df[, sortorder, names])`: Make a MultiIndex from a DataFrame.

MultiIndex properties

- `MultiIndex.names`: Names of levels in MultiIndex.
- `MultiIndex.levels`
- `MultiIndex.codes`
- `MultiIndex.nlevels`: Integer number of levels in this MultiIndex.
- `MultiIndex.levshape`: A tuple with the length of each level.
- `MultiIndex.dtypes`: Return the dtypes as a Series for the underlying MultiIndex.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.MultiIndex.levels

MultiIndex.levels

pandas.MultiIndex.codes

property MultiIndex.codes

pandas.MultiIndex.dtypes

MultiIndex.dtypes
  Return the dtypes as a Series for the underlying MultiIndex.

MultiIndex components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiIndex.set_levels</td>
<td>Set new levels on MultiIndex.</td>
</tr>
<tr>
<td>MultiIndex.set_codes</td>
<td>Set new codes on MultiIndex.</td>
</tr>
<tr>
<td>MultiIndex.to_flat_index()</td>
<td>Convert a MultiIndex to an Index of Tuples containing the level values.</td>
</tr>
<tr>
<td>MultiIndex.to_frame([index, name])</td>
<td>Create a DataFrame with the levels of the MultiIndex as columns.</td>
</tr>
<tr>
<td>MultiIndex.sortlevel([level, ascending, ...])</td>
<td>Sort MultiIndex at the requested level.</td>
</tr>
<tr>
<td>MultiIndex.droplevel([level])</td>
<td>Return MultiIndex at the requested level.</td>
</tr>
<tr>
<td>MultiIndex.swaplevel([i, j])</td>
<td>Swap level i with level j.</td>
</tr>
<tr>
<td>MultiIndex.reorder_levels(order)</td>
<td>Rearrange levels using input order.</td>
</tr>
<tr>
<td>MultiIndex.remove_unused_levels()</td>
<td>Create new MultiIndex from current that removes unused levels.</td>
</tr>
</tbody>
</table>

MultiIndex selecting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiIndex.get_loc(key[, method])</td>
<td>Get location for a label or a tuple of labels.</td>
</tr>
<tr>
<td>MultiIndex.get_locs(seq)</td>
<td>Get location for a sequence of labels.</td>
</tr>
<tr>
<td>MultiIndex.get_loc_level(key[, level, ...])</td>
<td>Get location and sliced index for requested label(s)/level(s).</td>
</tr>
<tr>
<td>MultiIndex.get_indexer(target[, method, ...])</td>
<td>Compute indexer and mask for new index given the current index.</td>
</tr>
<tr>
<td>MultiIndex.get_level_values(level)</td>
<td>Return vector of label values for requested level.</td>
</tr>
</tbody>
</table>
pandas.MultiIndex.get_loc

MultiIndex.get_loc(key, method=None)
Get location for a label or a tuple of labels.

The location is returned as an integer/slice or boolean mask.

Parameters
key [label or tuple of labels (one for each level)]
method [None]

Returns
loc [int, slice object or boolean mask] If the key is past the lexsort depth, the return may be a
boolean mask array, otherwise it is always a slice or int.

See also:
Index.get_loc The get_loc method for (single-level) index.
MultiIndex.slice_locs Get slice location given start label(s) and end label(s).
MultiIndex.get_locs Get location for a label/slice/list/mask or a sequence of such.

Notes

The key cannot be a slice, list of same-level labels, a boolean mask, or a sequence of such. If you want to use
those, use MultiIndex.get_locs() instead.

Examples

```python
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
>>> mi.get_loc('b')
slice(1, 3, None)

>>> mi.get_loc(('b', 'e'))
1
```

pandas.MultiIndex.get_loc_level

MultiIndex.get_loc_level(key, level=0, drop_level=True)
Get location and sliced index for requested label(s)/level(s).

Parameters
key [label or sequence of labels]
level [int/level name or list thereof, optional]
drop_level [bool, default True] If False, the resulting index will not drop any level.

Returns
loc [A 2-tuple where the elements are:] Element 0: int, slice object or boolean array Element
1: The resulting sliced multiindex/index. If the key contains all levels, this will be None.

See also:
**MultiIndex.get_loc** Get location for a label or a tuple of labels.

**MultiIndex.get_locs** Get location for a label/slice/list/mask or a sequence of such.

**Examples**

```python
cpy = pd.MultiIndex.from_arrays([list('abb'), list('def')], names=['A', 'B'])

>>> mi.get_loc_level('b')(slice(1, 3, None), Index(['e', 'f'], dtype='object', name='B'))
```

```python
cpy = pd.MultiIndex.from_arrays([list('abb'), list('def')], names=['A', 'B'])

>>> mi.get_loc_level('e', level='B')(array([False, True, False]), Index(['b'], dtype='object', name='A'))
```

```python
cpy = pd.MultiIndex.from_arrays([list('abb'), list('def')], names=['A', 'B'])

>>> mi.get_loc_level(['b', 'e'])
(1, None)
```

**pandas.MultiIndex.get_indexer**

**MultiIndex.get_indexer**(target, method=None, limit=None, tolerance=None)  
Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

**Parameters**

- **target** [Index]  
- **method** [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]  
  - default: exact matches only.  
  - pad / ffill: find the PREVIOUS index value if no exact match.  
  - backfill / bfill: use NEXT index value if no exact match  
  - nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.  
- **limit** [int, optional] Maximum number of consecutive labels in target to match for inexact matches.  
- **tolerance** [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation abs(index[indexer] - target) <= tolerance.  
  Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index’s type.

**Returns**

- **indexer** [np.ndarray[np.intp]] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.
Notes

Returns -1 for unmatched values, for further explanation see the example below.

Examples

```python
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

**pandas.MultiIndex.get_level_values**

MultiIndex.get_level_values(level)

Return vector of label values for requested level.

Length of returned vector is equal to the length of the index.

Parameters

- **level** [int or str] level is either the integer position of the level in the MultiIndex, or the name of the level.

Returns

- **values** [Index] Values is a level of this MultiIndex converted to a single Index (or subclass thereof).

Notes

If the level contains missing values, the result may be casted to float with missing values specified as NaN. This is because the level is converted to a regular Index.

Examples

Create a MultiIndex:

```python
>>> mi = pd.MultiIndex.from_arrays((list('abc'), list('def')))
>>> mi.names = ['level_1', 'level_2']
```

Get level values by supplying level as either integer or name:

```python
>>> mi.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object', name='level_1')
>>> mi.get_level_values('level_2')
Index(['d', 'e', 'f'], dtype='object', name='level_2')
```

If a level contains missing values, the return type of the level maybe casted to float.

```python
>>> pd.MultiIndex.from_arrays([[1, None, 2], [3, 4, 5]]).dtypes
level_0    int64
level_1    int64
```

(continues on next page)
dtype: object

```python
>>> pd.MultiIndex.from_arrays([[1, None, 2], [3, 4, 5]]).get_level_values(0)
Float64Index([1.0, nan, 2.0], dtype='float64')
```

### 3.6.6 DatetimeIndex

**datetime64**

```
datetime64([data, freq, tz, normalize, ...]) Immutable ndarray-like of datetime64 data.
```

**pandas.DatetimeIndex**

```python
class pandas.DatetimeIndex(data=None, freq=NoDefault.no_default, tz=None, normalize=False,
closed=None, ambiguous='raise', dayfirst=False, yearfirst=False, dtype=None,
copy=False, name=None)
```

Immutable ndarray-like of datetime64 data.

Represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata.

**Parameters**

- `data` [array-like (1-dimensional), optional] Optional datetime-like data to construct index with.
- `freq` [str or pandas offset object, optional] One of pandas date offset strings or corresponding objects. The string ‘infer’ can be passed in order to set the frequency of the index as the inferred frequency upon creation.
- `tz` [pytz.timezone or dateutil.tz.tzfile or datetime.tzinfo or str] Set the Timezone of the data.
- `normalize` [bool, default False] Normalize start/end dates to midnight before generating date range.
- `closed` [‘left’, ‘right’], optional] Set whether to include start and end that are on the boundary. The default includes boundary points on either end.
- `ambiguous` [‘infer’, bool-ndaray, ‘NaT’, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the ambiguous parameter dictates how ambiguous times should be handled.
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-narray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
- `dayfirst` [bool, default False] If True, parse dates in data with the day first order.
- `yearfirst` [bool, default False] If True parse dates in data with the year first order.
- `dtype` [numpy.dtype or DatetimeTZDtype or str, default None] Note that the only NumPy dtype allowed is ‘datetime64[ns]’.
- `copy` [bool, default False] Make a copy of input ndarray.
name  [label, default None] Name to be stored in the index.

See also:

- Index  The base pandas Index type.
- TimedeltaIndex  Index of timedelta64 data.
- PeriodIndex  Index of Period data.
- to_datetime  Convert argument to datetime.
- date_range  Create a fixed-frequency DatetimeIndex.

Notes

To learn more about the frequency strings, please see this link.
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>year</code></td>
<td>The year of the datetime.</td>
</tr>
<tr>
<td><code>month</code></td>
<td>The month as January=1, December=12.</td>
</tr>
<tr>
<td><code>day</code></td>
<td>The day of the datetime.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>The hours of the datetime.</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>The minutes of the datetime.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>The seconds of the datetime.</td>
</tr>
<tr>
<td><code>microsecond</code></td>
<td>The microseconds of the datetime.</td>
</tr>
<tr>
<td><code>nanosecond</code></td>
<td>The nanoseconds of the datetime.</td>
</tr>
<tr>
<td><code>date</code></td>
<td>Returns numpy array of python <code>datetime.date</code> objects.</td>
</tr>
<tr>
<td><code>time</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects.</td>
</tr>
<tr>
<td><code>timetz</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects with timezone information.</td>
</tr>
<tr>
<td><code>dayofyear</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>day_of_year</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>weekofyear</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>week</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>day_of_week</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>weekday</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>quarter</code></td>
<td>The quarter of the date.</td>
</tr>
<tr>
<td><code>tz</code></td>
<td>Return the timezone.</td>
</tr>
<tr>
<td><code>freq</code></td>
<td>Return the frequency object if it is set, otherwise None.</td>
</tr>
<tr>
<td><code>freqstr</code></td>
<td>Return the frequency object as a string if its set, otherwise None.</td>
</tr>
<tr>
<td><code>is_month_start</code></td>
<td>Indicates whether the date is the first day of the month.</td>
</tr>
<tr>
<td><code>is_month_end</code></td>
<td>Indicates whether the date is the last day of the month.</td>
</tr>
<tr>
<td><code>is_quarter_start</code></td>
<td>Indicator for whether the date is the first day of a quarter.</td>
</tr>
<tr>
<td><code>is_quarter_end</code></td>
<td>Indicator for whether the date is the last day of a quarter.</td>
</tr>
<tr>
<td><code>is_year_start</code></td>
<td>Indicate whether the date is the first day of a year.</td>
</tr>
<tr>
<td><code>is_year_end</code></td>
<td>Indicate whether the date is the last day of the year.</td>
</tr>
<tr>
<td><code>is_leap_year</code></td>
<td>Boolean indicator if the date belongs to a leap year.</td>
</tr>
<tr>
<td><code>inferred_freq</code></td>
<td>Tries to return a string representing a frequency guess, generated by infer_freq.</td>
</tr>
</tbody>
</table>
pandas.DatetimeIndex.year

**property DatetimeIndex.year**

The year of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="Y")
... )
>>> datetime_series
datetime_series: 2000-12-31 2001-12-31 2002-12-31
0 2000
1 2001
2 2002
dtype: datetime64[ns]

>>> datetime_series.dt.year
0 2000
1 2001
2 2002
dtype: int64
```

pandas.DatetimeIndex.month

**property DatetimeIndex.month**

The month as January=1, December=12.

**Examples**

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="M")
... )
>>> datetime_series
0 2000-01-31
1 2000-02-29
2 2000-03-31
dtype: datetime64[ns]

>>> datetime_series.dt.month
0 1
1 2
2 3
dtype: int64
```
**pandas.DatetimeIndex.day**

**property DatetimeIndex.day**

The day of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="D")
... )
>>> datetime_series
0 2000-01-01
1 2000-01-02
2 2000-01-03
dtype: datetime64[ns]
>>> datetime_series.dt.day
0 1
1 2
2 3
dtype: int64
```

**pandas.DatetimeIndex.hour**

**property DatetimeIndex.hour**

The hours of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
...     pd.date_range("2000-01-01", periods=3, freq="h")
... )
>>> datetime_series
0 2000-01-01 00:00:00
1 2000-01-01 01:00:00
2 2000-01-01 02:00:00
dtype: datetime64[ns]
>>> datetime_series.dt.hour
0 0
1 1
2 2
dtype: int64
```
pandas.DatetimeIndex.minute

**property** DatetimeIndex.minute

The minutes of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
    ...     pd.date_range("2000-01-01", periods=3, freq="T")
    ... )
>>> datetime_series
0  2000-01-01 00:00:00
1  2000-01-01 00:01:00
2  2000-01-01 00:02:00
dtype: datetime64[ns]
>>> datetime_series.dt.minute
0  0
1  1
2  2
dtype: int64
```

pandas.DatetimeIndex.second

**property** DatetimeIndex.second

The seconds of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
    ...     pd.date_range("2000-01-01", periods=3, freq="s")
    ... )
>>> datetime_series
0  2000-01-01 00:00:00
1  2000-01-01 00:00:01
2  2000-01-01 00:00:02
dtype: datetime64[ns]
>>> datetime_series.dt.second
0  0
1  1
2  2
dtype: int64
```
pandas.DatetimeIndex.microsecond

**property** DatetimeIndex.microsecond

The microseconds of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
    ...   pd.date_range("2000-01-01", periods=3, freq="us")
    ... )
>>> datetime_series
0 2000-01-01 00:00:00.000000
1 2000-01-01 00:00:00.000001
2 2000-01-01 00:00:00.000002
dtype: datetime64[ns]
>>> datetime_series.dt.microsecond
0 0
1 1
2 2
dtype: int64
```

pandas.DatetimeIndex.nanosecond

**property** DatetimeIndex.nanosecond

The nanoseconds of the datetime.

**Examples**

```python
>>> datetime_series = pd.Series(
    ...   pd.date_range("2000-01-01", periods=3, freq="ns")
    ... )
>>> datetime_series
0 2000-01-01 00:00:00.000000
1 2000-01-01 00:00:00.000001
2 2000-01-01 00:00:00.000002
dtype: datetime64[ns]
>>> datetime_series.dt.nanosecond
0 0
1 1
2 2
dtype: int64
```
pandas.DatetimeIndex.date

**property** DatetimeIndex.date
Returns numpy array of python `datetime.date` objects.
Namely, the date part of Timestamps without time and timezone information.

pandas.DatetimeIndex.time

**property** DatetimeIndex.time
Returns numpy array of `datetime.time` objects.
The time part of the Timestamps.

pandas.DatetimeIndex.timetz

**property** DatetimeIndex.timetz
Returns numpy array of `datetime.time` objects with timezone information.
The time part of the Timestamps.

pandas.DatetimeIndex.dayofyear

**property** DatetimeIndex.dayofyear
The ordinal day of the year.

pandas.DatetimeIndex.day_of_year

**property** DatetimeIndex.day_of_year
The ordinal day of the year.

pandas.DatetimeIndex.weekofyear

**property** DatetimeIndex.weekofyear
The week ordinal of the year.
Deprecated since version 1.1.0.
weekofyear and week have been deprecated. Please use DatetimeIndex.isocalendar().week instead.
**pandas.DatetimeIndex.week**

**property** DatetimeIndex.week  
The week ordinal of the year.  
Deprecated since version 1.1.0.  
weekofyear and week have been deprecated. Please use DatetimeIndex.isocalendar().week instead.

**pandas.DatetimeIndex.dayofweek**

**property** DatetimeIndex.dayofweek  
The day of the week with Monday=0, Sunday=6.  
Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the dt accessor) or DatetimeIndex.

**Returns**  
*Series or Index*  Containing integers indicating the day number.

**See also:**

*Series.dt.dayofweek*  Alias.  
*Series.dt.weekday*  Alias.  
*Series.dt.day_name*  Returns the name of the day of the week.

**Examples**

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31   5
2017-01-01   6
2017-01-02   0
2017-01-03   1
2017-01-04   2
2017-01-05   3
2017-01-06   4
2017-01-07   5
2017-01-08   6
Freq: D, dtype: int64
```
pandas.DatetimeIndex.day_of_week

**property DatetimeIndex.day_of_week**

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the `dt` accessor) or DatetimeIndex.

Returns

Series or Index  Containing integers indicating the day number.

See also:

Series.dt.dayofweek  Alias.

Series.dt.weekday  Alias.

Series.dt.day_name  Returns the name of the day of the week.

Examples

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31   5
2017-01-01   6
2017-01-02   0
2017-01-03   1
2017-01-04   2
2017-01-05   3
2017-01-06   4
2017-01-07   5
2017-01-08   6
Freq: D, dtype: int64
```

pandas.DatetimeIndex.weekday

**property DatetimeIndex.weekday**

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the `dt` accessor) or DatetimeIndex.

Returns

Series or Index  Containing integers indicating the day number.

See also:

Series.dt.dayofweek  Alias.

Series.dt.weekday  Alias.

Series.dt.day_name  Returns the name of the day of the week.
Examples

```python
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31 5
2017-01-01 6
2017-01-02 0
2017-01-03 1
2017-01-04 2
2017-01-05 3
2017-01-06 4
2017-01-07 5
2017-01-08 6
Freq: D, dtype: int64
```

**pandas.DatetimeIndex.quarter**

**property** `DatetimeIndex.quarter`  
The quarter of the date.

**pandas.DatetimeIndex.tz**

**property** `DatetimeIndex.tz`  
Return the timezone.

Returns  
```
datetime.tzinfo, pytz.tzinfo.BaseTZInfo, dateutil.tz.tzfile, or None  
Returns None  
when the array is tz-naive.
```

**pandas.DatetimeIndex.freq**

**property** `DatetimeIndex.freq`  
Return the frequency object if it is set, otherwise None.

**pandas.DatetimeIndex.freqstr**

**property** `DatetimeIndex.freqstr`  
Return the frequency object as a string if its set, otherwise None.
pandas.DatetimeIndex.is_month_start

**property** DatetimeIndex.is_month_start

Indicates whether the date is the first day of the month.

**Returns**

*Series or array* For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

**See also:**

*is_month_start* Return a boolean indicating whether the date is the first day of the month.

*is_month_end* Return a boolean indicating whether the date is the last day of the month.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0  2018-02-27
1  2018-02-28
2  2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0   False
1   False
2    True
dtype: bool
>>> s.dt.is_month_end
0   False
1    True
2   False
dtype: bool
```

```python
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```
**pandas.DatetimeIndex.is_month_end**

**property DatetimeIndex.is_month_end**
Indicates whether the date is the last day of the month.

**Returns**
- **Series or array** For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

**See also:**
- **is_month_start** Return a boolean indicating whether the date is the first day of the month.
- **is_month_end** Return a boolean indicating whether the date is the last day of the month.

**Examples**
This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0  2018-02-27
1  2018-02-28
2  2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0   False
1   False
2    True
dtype: bool
>>> s.dt.is_month_end
0   False
1    True
2   False
dtype: bool
```

```python
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx
```

```python
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```
pandas.DatetimeIndex.is_quarter_start

**property** `DatetimeIndex.is_quarter_start`

Indicator for whether the date is the first day of a quarter.

**Returns**

`is_quarter_start` [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

`quarter` Return the quarter of the date.

`is_quarter_end` Similar property for indicating the quarter start.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",...periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,...is_quarter_start=df.dates.dt.is_quarter_start)
dates   quarter   is_quarter_start
0 2017-03-30     1        False
1 2017-03-31     1        False
2 2017-04-01     2         True
3 2017-04-02     2        False
```

```python
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(["2017-03-30", '2017-03-31', '2017-04-01', '2017-04-02'],...dtype='datetime64[ns]', freq='D')
```

```python
>>> idx.is_quarter_start
array([False, False,   True, False])
```

pandas.DatetimeIndex.is_quarter_end

**property** `DatetimeIndex.is_quarter_end`

Indicator for whether the date is the last day of a quarter.

**Returns**

`is_quarter_end` [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

`quarter` Return the quarter of the date.
**is_quarter_start**  Similar property indicating the quarter start.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> df = pd.DataFrame({'dates': pd.date_range('2017-03-30', periods=4), ...
                    'is_quarter_start': df.dates.dt.is_quarter_start, ...
                    'is_quarter_end': df.dates.dt.is_quarter_end})
dates    quarter    is_quarter_end
0 2017-03-30    1      False
1 2017-03-31    1      True
2 2017-04-01    2      False
3 2017-04-02    2      False
```

```python
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx.is_quarter_end
array([False, True, False, False])
```

### pandas.DatetimeIndex.is_year_start

**property** `DatetimeIndex.is_year_start`

Indicate whether the date is the first day of a year.

**Returns**

- **Series or DatetimeIndex** The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

- **is_year_end**  Similar property indicating the last day of the year.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```python
>>> dates = pd.Series(pd.date_range('2017-12-30', periods=3))
>>> dates
0  2017-12-30
1  2017-12-31
2  2018-01-01
dtype: datetime64[ns]
```
```python
>>> dates.dt.is_year_start
0   False
1   False
2    True
dtype: bool

>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
dtype='datetime64[ns]', freq='D')

>>> idx.is_year_start
array([False, False,  True])

pandas.DatetimeIndex.is_year_end

property DatetimeIndex.is_year_end
Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex  The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start  Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> idx.is_year_end
array([False,  True, False])
```

**pandas.DatetimeIndex.is_leap_year**

**property** DatetimeIndex.is_leap_year

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

**Returns**

Series or ndarray  Booleans indicating if dates belong to a leap year.

**Examples**

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```python
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(["2012-12-31", "2013-12-31", "2014-12-31"],
               dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False])
```

```python
>>> dates_series = pd.Series(idx)
>>> dates_series
0 2012-12-31
1 2013-12-31
2 2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0   True
1  False
2  False
dtype: bool
```

**pandas.DatetimeIndex.inferred_freq**

DatetimeIndex.inferred_freq

Tries to return a string representing a frequency guess, generated by infer_freq. Returns None if it can't autodetect the frequency.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>normalize(*args, **kwargs)</code></td>
<td>Convert times to midnight. The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.</td>
</tr>
<tr>
<td><code>strftime(*args, **kwargs)</code></td>
<td>Convert to Index using specified date_format.</td>
</tr>
<tr>
<td><code>snap(freq)</code></td>
<td>Snap time stamps to nearest occurring frequency.</td>
</tr>
<tr>
<td><code>tz_convert(tz)</code></td>
<td>Convert tz-aware Datetime Array/Index from one time zone to another.</td>
</tr>
<tr>
<td><code>tz_localize(tz[, ambiguous, nonexistent])</code></td>
<td>Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.</td>
</tr>
<tr>
<td><code>round(*args, **kwargs)</code></td>
<td>Perform round operation on the data to the specified freq.</td>
</tr>
<tr>
<td><code>floor(*args, **kwargs)</code></td>
<td>Perform floor operation on the data to the specified freq.</td>
</tr>
<tr>
<td><code>ceil(*args, **kwargs)</code></td>
<td>Perform ceil operation on the data to the specified freq.</td>
</tr>
<tr>
<td><code>to_period(*args, **kwargs)</code></td>
<td>Cast to PeriodArray/Index at a particular frequency.</td>
</tr>
<tr>
<td><code>to_perioddelta(freq)</code></td>
<td>Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq.</td>
</tr>
<tr>
<td><code>to_pydatetime(*args, **kwargs)</code></td>
<td>Return Datetime Array/Index as object ndarray of datetime.datetime objects.</td>
</tr>
<tr>
<td><code>to_series([keep_tz, index, name])</code></td>
<td>Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.</td>
</tr>
<tr>
<td><code>to_frame([index, name])</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td><code>month_name(*args, **kwargs)</code></td>
<td>Return the month names of the DateTimeIndex with specified locale.</td>
</tr>
<tr>
<td><code>day_name(*args, **kwargs)</code></td>
<td>Return the day names of the DateTimeIndex with specified locale.</td>
</tr>
<tr>
<td><code>mean(*args, **kwargs)</code></td>
<td>Return the mean value of the Array.</td>
</tr>
<tr>
<td><code>std(*args, **kwargs)</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
</tbody>
</table>

### `pandas.DatetimeIndex.normalize`

**DatetimeIndex.normalize(*args, **kwargs)**

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datatime values under the `.dt` accessor, and directly on Datetime Array/Index.

#### Returns

- **DatetimeArray, DatetimeIndex or Series** The same type as the original data. Series will have the same name and index. DatetimeIndex will have the same name.

#### See also:

- **floor** Floor the datetimes to the specified freq.
ceil  Ceil the datetimes to the specified freq.

round  Round the datetimes to the specified freq.

Examples

```python
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
                      ...       periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

pandas.DatetimeIndex.strftime

DatetimeIndex.strftime(*args, **kwargs)

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

Parameters

date_format [str] Date format string (e.g. “%Y-%m-%d”).

Returns

ndarray[object] NumPy ndarray of formatted strings.

See also:

to_datetime  Convert the given argument to datetime.

DatetimeIndex.normalize  Return DatetimeIndex with times to midnight.

DatetimeIndex.round  Round the DatetimeIndex to the specified freq.

DatetimeIndex.floor  Floor the DatetimeIndex to the specified freq.

Examples

```python
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
                      ...       periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```
**pandas.DatetimeIndex.snap**

```python
DatetimeIndex.snap(freq='S')
```

Snap time stamps to nearest occurring frequency.

**Returns**

`DatetimeIndex`

**pandas.DatetimeIndex.tz_convert**

```python
DatetimeIndex.tz_convert(tz)
```

Convert tz-aware Datetime Array/Index from one time zone to another.

**Parameters**

- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A `tz` of None will convert to UTC and remove the timezone information.

**Returns**

`Array or Index`

**Raises**

`TypeError` If Datetime Array/Index is tz-naive.

**See also:**

- `DatetimeIndex.tz` A timezone that has a variable offset from UTC.
- `DatetimeIndex.tz_localize` Localize tz-naive DatetimeIndex to a given time zone, or remove time-zone from a tz-aware DatetimeIndex.

**Examples**

With the `tz` parameter, we can change the DatetimeIndex to other time zones:

```python
>>> dti = pd.date_range(start='2014-08-01 09:00',...
... freq='H', periods=3, tz='Europe/Berlin')
```

```python
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
  '2014-08-01 10:00:00+02:00',
  '2014-08-01 11:00:00+02:00'],
  dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```python
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
  '2014-08-01 03:00:00-05:00',
  '2014-08-01 04:00:00-05:00'],
  dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> dti = pd.date_range(start='2014-08-01 09:00', freq='H', periods=3, tz='Europe/Berlin')

>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
               dtype='datetime64[ns, Europe/Berlin]', freq='H')

>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
               dtype='datetime64[ns]', freq='H')
```

**pandas.DatetimeIndex.tz_localize**

`DatetimeIndex.tz_localize(tz, ambiguous='raise', nonexistent='raise')`

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone.

This method can also be used to do the inverse – to create a time zone unaware object from an aware object. To that end, pass `tz=None`.

**Parameters**

- `tz` [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

- `ambiguous` ['infer', ‘NaT’, bool array, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the `ambiguous` parameter dictates how ambiguous times should be handled.
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

- `nonexistent` ['shift_forward', ‘shift_backward’, ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return NaT where there are nonexistent times
• timedelta objects will shift nonexistent times by the timedelta
• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

**Same type as self** Array/Index converted to the specified time zone.

**Raises**

**TypeError** If the Datetime Array/Index is tz-aware and tz is not None.

**See also:**

*DatetimeIndex.tz_convert* Convert tz-aware DatetimeIndex from one time zone to another.

**Examples**

```python
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'], dtype='datetime64[ns]', freq='D')

Localize DatetimeIndex in US/Eastern time zone:

```python
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
              '2018-03-02 09:00:00-05:00',
              '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)

With the tz=None, we can remove the time zone information while keeping the local time (not converted to UTC):

```python
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq=None)

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```python
>>> s = pd.to_datetime(pd.Series(['2018-10-28 01:30:00',
                                ...                     '2018-10-28 02:00:00',
                                ...                     '2018-10-28 02:30:00',
                                ...                     '2018-10-28 03:00:00',
                                ...                     '2018-10-28 03:30:00']))
>>> s.dt.tz_localize('CET', ambiguous='infer')
0 2018-10-28 01:30:00+02:00
1 2018-10-28 02:00:00+02:00
2 2018-10-28 02:30:00+02:00
3 2018-10-28 03:00:00+01:00
4 2018-10-28 03:30:00+01:00
```
In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly:

```python
>>> s = pd.to_datetime(pd.Series(['2018-10-28 01:20:00',
                                '2018-10-28 02:36:00',
                                '2018-10-28 03:46:00']))
>>> s.dt.tz_localize('CET', ambiguous=np.array([True, True, False]))
0  2018-10-28 01:20:00+02:00
1  2018-10-28 02:36:00+02:00
2  2018-10-28 03:46:00+01:00
dtype: datetime64[ns, CET]
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or ‘shift_forward’ or ‘shift_backwards’:

```python
>>> s = pd.to_datetime(pd.Series(['2015-03-29 02:30:00',
                                '2015-03-29 03:30:00']))
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
0  2015-03-29 03:00:00+02:00
1  2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]

>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
0  2015-03-29 01:59:59.999999999+01:00
1  2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]

>>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
0  2015-03-29 03:30:00+02:00
1  2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]
```

**pandas.DatetimeIndex.round**

DatetimeIndex.round(*args, **kwargs)

Perform round operation on the data to the specified freq.

**Parameters**

- **freq** [str or Offset] The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.

- **ambiguous** ['infer', bool-ndarray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time. False designates a non-DST time (note that this flag is only applicable for ambiguous times)
• ‘NaT’ will return NaT where there are ambiguous times
• ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

**nonexistent** ['shift_forward', 'shift_backward', 'NaT', timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
• ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
• ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
• ‘NaT’ will return NaT where there are nonexistent times
• timedelta objects will shift nonexistent times by the timedelta
• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

**DatetimeIndex, TimedeltaIndex, or Series** Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

**ValueError** if the freq cannot be converted.

**Notes**

If the timestamps have a timezone, rounding will take place relative to the local (“wall”) time and re-localized to the same timezone. When rounding near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

**Examples**

**DatetimeIndex**

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'], dtype='datetime64[ns]', freq='T')

>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
               dtype='datetime64[ns]', freq=None)
```

**Series**

```python
>>> pd.Series(rng).dt.round("H")
0  2018-01-01 12:00:00
1  2018-01-01 12:00:00
2  2018-01-01 12:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 03:30:00'], tz="Europe/Amsterdam")
>>> rng_tz.floor("2H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

```python
>>> rng_tz.floor("2H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

**pandas.DatetimeIndex.floor**

**DatetimeIndex.floor**(\*\*args, \*\*kwargs)

Perform floor operation on the data to the specified \textit{freq}.

**Parameters**

- **freq** [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See \textit{frequency aliases} for a list of possible \textit{freq} values.

- **ambiguous** ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return \textit{NaT} where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

- **nonexistent** ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return \textit{NaT} where there are nonexistent times
  - timedelta objects will shift nonexistent times by the timedelta
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

- \textit{DatetimeIndex}, \textit{TimedeltaIndex}, or \textit{Series} Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

- ValueError if the \textit{freq} cannot be converted.
Notes

If the timestamps have a timezone, flooring will take place relative to the local (“wall”) time and re-localized to the same timezone. When flooring near daylight savings time, use `nonexistent` and `ambiguous` to control the re-localization behavior.

Examples

**Datet imeIndex**

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'], dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'], dtype='datetime64[ns]', freq=None)
```

**Series**

```python
>>> pd.Series(rng).dt.floor("H")
0    2018-01-01 11:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use `ambiguous` or `nonexistent` to control how the timestamp should be re-localized.

```python
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 03:30:00'], tz="Europe/Amsterdam")
>>> rng_tz.floor("2H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
>>> rng_tz.floor("2H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

**pandas.Datet imeIndex.ceil**

```python
Datet imeIndex.ceil(*args, **kwargs)
```

Perform ceil operation on the data to the specified `freq`.

Parameters

- `freq` [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See `frequency aliases` for a list of possible `freq` values.
**ambiguous** ['infer', bool-ndarray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:

- ‘infer’ will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- ‘NaT’ will return NaT where there are ambiguous times
- ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

**nonexistent** ['shift_forward', 'shift_backward', ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
- ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
- ‘NaT’ will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- ‘raise’ will raise a NonExistentTimeError if there are nonexistent times.

**Returns**

DatetimeIndex, TimedeltaIndex, or Series  Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

ValueError if the freq cannot be converted.

**Notes**

If the timestamps have a timezone, ceiling will take place relative to the local (“wall”) time and re-localized to the same timezone. When ceiling near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

**Examples**

**DatetimeIndex**

```python
g = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
g
```

```
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00', '2018-01-01 12:01:00'],
             dtype='datetime64[ns]', freq='T')
```

```python
g.ceil('H')
g
```

```
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00', '2018-01-01 13:00:00'],
                dtype='datetime64[ns]', freq=None)
```

**Series**
When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 01:30:00'], tz='Europe/Amsterdam')

>>> rng_tz.ceil('H', ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
          dtype='datetime64[ns, Europe/Amsterdam]', freq=None)

>>> rng_tz.ceil('H', ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
          dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

**pandas.DatetimeIndex.to_period**

`DatetimeIndex.to_period(*args, **kwargs)`

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

**Parameters**

freq [str or Offset, optional] One of pandas' *offset strings* or an Offset object. Will be inferred by default.

**Returns**

PeriodArray/Index

**Raises**

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

**See also:**

**PeriodIndex** Immutable ndarray holding ordinal values.

**DatetimeIndex.to_pydatetime** Return DatetimeIndex as object.
Examples

```python
>>> df = pd.DataFrame({'y': [1, 2, 3]},
                   index=pd.date_range('2000-03-31 00:00:00', '2000-08-31 00:00:00', periods=3))
>>> df.index.to_period('M')
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]')

Infer the daily frequency

```python
>>> idx = pd.date_range('2017-01-01', periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]')
```

**pandas.DatetimeIndex.to_perioddelta**

`DatetimeIndex.to_perioddelta(freq)`

Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq. Used for vectorized offsets.

**Parameters**

- `freq` [Period frequency]

**Returns**

TimedeltaArray/Index

**pandas.DatetimeIndex.to_pydatetime**

`DatetimeIndex.to_pydatetime(*args, **kwargs)`

Return Datetime Array/Index as object ndarray of datetime.datetime objects.

**Returns**

- `datetimes` [ndarray[object]]

**pandas.DatetimeIndex.to_series**

`DatetimeIndex.to_series(keep_tz=NoDefault.no_default, index=None, name=None)`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.

**Parameters**

- `keep_tz` [optional, defaults True] Return the data keeping the timezone.

  If `keep_tz` is True:

  If the timezone is not set, the resulting Series will have a datetime64[ns] dtype.
Otherwise the Series will have an datetime64[ns, tz] dtype; the tz will be preserved.

If keep_tz is False:
Series will have a datetime64[ns] dtype. TZ aware objects will have the tz removed.

Changed in version 1.0.0: The default value is now True. In a future version, this keyword will be removed entirely. Stop passing the argument to obtain the future behavior and silence the warning.

- **index** [Index, optional] Index of resulting Series. If None, defaults to original index.
- **name** [str, optional] Name of resulting Series. If None, defaults to name of original index.

**Returns**
Series

**pandas.DatetimeIndex.to_frame**

**DatetimeIndex.to_frame**(index=True, name=NoDefault.no_default)
Create a DataFrame with a column containing the Index.

**Parameters**
- **index** [bool, default True] Set the index of the returned DataFrame as the original Index.
- **name** [object, default None] The passed name should substitute for the index name (if it has one).

**Returns**
DataFrame DataFrame containing the original Index data.

**See also:**
- **Index.to_series** Convert an Index to a Series.
- **Series.to_frame** Convert Series to DataFrame.

**Examples**

```python
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
animal
animal
Ant   Ant
Bear  Bear
Cow   Cow
```

By default, the original Index is reused. To enforce a new Index:

```python
>>> idx.to_frame(index=False)
animal
0   Ant
```

(continues on next page)
To override the name of the resulting column, specify `name`:

```python
>>> idx.to_frame(index=False, name='zoo')
   zoo
0  Ant
1  Bear
2  Cow
```

### pandas.DatetimeIndex.month_name

**DatetimeIndex.month_name(*args, **kwargs)**

Return the month names of the DateTimeIndex with specified locale.

**Parameters**

- `locale` [str, optional] Locale determining the language in which to return the month name. Default is English locale.

**Returns**

- `Index` Index of month names.

**Examples**

```python
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
```

```python
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'], dtype='datetime64[ns]', freq='M')
```

```python
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

### pandas.DatetimeIndex.day_name

**DatetimeIndex.day_name(*args, **kwargs)**

Return the day names of the DateTimeIndex with specified locale.

**Parameters**

- `locale` [str, optional] Locale determining the language in which to return the day name. Default is English locale.

**Returns**

- `Index` Index of day names.

```python
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
```

```python
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'], dtype='datetime64[ns]', freq='M')
```

```python
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```
Examples

```python
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
datetime64[ns]
Index(['2018-01-01', '2018-01-02', '2018-01-03'], dtype='datetime64[ns], freq='D')
```

```python
>>> idx.day_name()  
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

**pandas.DatetimeIndex.mean**

`DatetimeIndex.mean(*args, **kwargs)`

Return the mean value of the Array.

New in version 0.25.0.

**Parameters**

- `skipna` [bool, default True] Whether to ignore any NaT elements.
- `axis` [int, optional, default 0]

**Returns**

- `scalar` Timestamp or Timedelta.

**See also:**

- `numpy.ndarray.mean` Returns the average of array elements along a given axis.
- `Series.mean` Return the mean value in a Series.

**Notes**

mean is only defined for Datetime and Timedelta dtypes, not for Period.

**pandas.DatetimeIndex.std**

`DatetimeIndex.std(*args, **kwargs)`

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument.

**Parameters**

- `axis` [int optional, default None] Axis for the function to be applied on.
- `ddof` [int, default 1] Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- `skipna` [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

- `Timedelta`
## Time/Date Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DatetimeIndex.year</code></td>
<td>The year of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.month</code></td>
<td>The month as January=1, December=12.</td>
</tr>
<tr>
<td><code>DatetimeIndex.day</code></td>
<td>The day of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.hour</code></td>
<td>The hours of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.minute</code></td>
<td>The minutes of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.second</code></td>
<td>The seconds of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.microsecond</code></td>
<td>The microseconds of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.nanosecond</code></td>
<td>The nanoseconds of the datetime.</td>
</tr>
<tr>
<td><code>DatetimeIndex.date</code></td>
<td>Returns numpy array of python <code>datetime.date</code> objects.</td>
</tr>
<tr>
<td><code>DatetimeIndex.time</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects.</td>
</tr>
<tr>
<td><code>DatetimeIndex.timetz</code></td>
<td>Returns numpy array of <code>datetime.time</code> objects with timezone information.</td>
</tr>
<tr>
<td><code>DatetimeIndex.dayofyear</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>DatetimeIndex.day_of_year</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>DatetimeIndex.weekofyear</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>DatetimeIndex.week</code></td>
<td>(DEPRECATED) The week ordinal of the year.</td>
</tr>
<tr>
<td><code>DatetimeIndex.dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>DatetimeIndex.day_of_week</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>DatetimeIndex.weekday</code></td>
<td>The day of the week.</td>
</tr>
<tr>
<td><code>DatetimeIndex.quarter</code></td>
<td>The quarter of the date.</td>
</tr>
<tr>
<td><code>DatetimeIndex.tz</code></td>
<td>Return the timezone.</td>
</tr>
<tr>
<td><code>DatetimeIndex.freq</code></td>
<td>Return the frequency object if it is set, otherwise None.</td>
</tr>
<tr>
<td><code>DatetimeIndex.freqstr</code></td>
<td>Return the frequency object as a string if its set, otherwise None.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_month_start</code></td>
<td>Indicates whether the date is the first day of the month.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_month_end</code></td>
<td>Indicates whether the date is the last day of the month.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_quarter_start</code></td>
<td>Indicator for whether the date is the first day of a quarter.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_quarter_end</code></td>
<td>Indicator for whether the date is the last day of a quarter.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_year_start</code></td>
<td>Indicate whether the date is the first day of a year.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_year_end</code></td>
<td>Indicate whether the date is the last day of the year.</td>
</tr>
<tr>
<td><code>DatetimexIndex.is_leap_year</code></td>
<td>Boolean indicator if the date belongs to a leap year.</td>
</tr>
<tr>
<td><code>DatetimexIndex.inferred_freq</code></td>
<td>Tries to return a string representing a frequency guess, generated by infer_freq.</td>
</tr>
</tbody>
</table>

## Selecting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DatetimeIndex.indexer_at_time(time[, asof])</code></td>
<td>Return index locations of values at particular time of day (e.g.</td>
</tr>
<tr>
<td><code>DatetimeIndex.indexer_between_time(...)[, ...]</code></td>
<td>Return index locations of values between particular times of day (e.g., 9:00-9:30AM).</td>
</tr>
</tbody>
</table>
pandas.DatetimeIndex.indexer_at_time

**DatetimeIndex.indexer_at_time**(time, asof=False)

Return index locations of values at particular time of day (e.g. 9:30AM).

**Parameters**

- **time** [datetime.time or str] Time passed in either as object (datetime.time) or as string in appropriate format ("%H:%M", "%H%M", "%I:%M%p", "%I%M%p", "%H:%M:%S", "%H%M%S", "%I:%M:%S%p", "%I%M%S%p").

**Returns**

np.ndarray[np.intp]

See also:

- **indexer_between_time** Get index locations of values between particular times of day.
- **DataFrame.at_time** Select values at particular time of day.

pandas.DatetimeIndex.indexer_between_time

**DatetimeIndex.indexer_between_time**(start_time, end_time, include_start=True, include_end=True)

Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

**Parameters**

- **start_time, end_time** [datetime.time or str] Time passed either as object (datetime.time) or as string in appropriate format ("%H:%M", "%H%M", "%I:%M%p", "%I%M%p", "%H:%M:%S", "%H%M%S", "%I:%M:%S%p", "%I%M%S%p").
- **include_start** [bool, default True]
- **include_end** [bool, default True]

**Returns**

np.ndarray[np.intp]

See also:

- **indexer_at_time** Get index locations of values at particular time of day.
- **DataFrame.between_time** Select values between particular times of day.

**Time-specific operations**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatetimeIndex.normalize(*args, **kwargs)</td>
<td>Convert times to midnight.</td>
</tr>
<tr>
<td>DatetimeIndex.strftime(*args, **kwargs)</td>
<td>Convert to Index using specified date_format.</td>
</tr>
<tr>
<td>DatetimeIndex.snap([freq])</td>
<td>Snap time stamps to nearest occurring frequency.</td>
</tr>
<tr>
<td>DatetimeIndex.tz_convert(tz)</td>
<td>Convert tz-aware Datetime Array/Index from one time zone to another.</td>
</tr>
<tr>
<td>DatetimeIndex.tz_localize([tz[, ambiguous, ...]])</td>
<td>Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.</td>
</tr>
<tr>
<td>DatetimeIndex.round(*args, **kwargs)</td>
<td>Perform round operation on the data to the specified freq.</td>
</tr>
<tr>
<td>DatetimeIndex.floor(*args, **kwargs)</td>
<td>Perform floor operation on the data to the specified freq.</td>
</tr>
<tr>
<td>DatetimeIndex.ceil(*args, **kwargs)</td>
<td>Perform ceil operation on the data to the specified freq.</td>
</tr>
<tr>
<td>DatetimeIndex.month_name(*args, **kwargs)</td>
<td>Return the month names of the DateTimeIndex with specified locale.</td>
</tr>
<tr>
<td>DatetimeIndex.day_name(*args, **kwargs)</td>
<td>Return the day names of the DateTimeIndex with specified locale.</td>
</tr>
</tbody>
</table>
Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DatetimeIndex.to_period(*args, **kwargs)</code></td>
<td>Cast to PeriodArray/Index at a particular frequency.</td>
</tr>
<tr>
<td><code>DatetimeIndex.to_perioddelta(freq)</code></td>
<td>Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq.</td>
</tr>
<tr>
<td><code>DatetimeIndex.to_pydatetime(*args, **kwargs)</code></td>
<td>Return Datetime Array/Index as object ndarray of datetime.datetime objects.</td>
</tr>
<tr>
<td><code>DatetimeIndex.to_series([keep_tz, index, name])</code></td>
<td>Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.</td>
</tr>
<tr>
<td><code>DatetimeIndex.to_frame([index, name])</code></td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DatetimeIndex.mean(*args, **kwargs)</code></td>
<td>Return the mean value of the Array.</td>
</tr>
<tr>
<td><code>DatetimeIndex.std(*args, **kwargs)</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
</tbody>
</table>

3.6.7 TimedeltaIndex

```
TimedeltaIndex([data, unit, freq, closed, ...])
```

Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects.

pandas.TimedeltaIndex

```
class pandas.TimedeltaIndex(data=None, unit=None, freq=NoDefault.no_default, closed=None,
    dtype=dtype('<m8[ns]'), copy=False, name=None)
```

Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects.

Parameters:

- **data** [array-like (1-dimensional), optional] Optional timedelta-like data to construct index with.
- **unit** [unit of the arg (D,h,m,s,ms,us,ns) denote the unit, optional] Which is an integer/float number.
- **freq** [str or pandas offset object, optional] One of pandas date offset strings or corresponding objects. The string ‘infer’ can be passed in order to set the frequency of the index as the inferred frequency upon creation.
- **copy** [bool] Make a copy of input ndarray.
- **name** [object] Name to be stored in the index.

See also:

- Index The base pandas Index type.
- Timedelta Represents a duration between two dates or times.
- DatetimeIndex Index of datetime64 data.
- PeriodIndex Index of Period data.
**timedelta_range**  Create a fixed-frequency TimedeltaIndex.

**Notes**

To learn more about the frequency strings, please see this link.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>days</td>
<td>Number of days for each element.</td>
</tr>
<tr>
<td>seconds</td>
<td>Number of seconds (&gt;= 0 and less than 1 day) for each element.</td>
</tr>
<tr>
<td>microseconds</td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td>nanoseconds</td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</td>
</tr>
<tr>
<td>components</td>
<td>Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.</td>
</tr>
<tr>
<td>inferred_freq</td>
<td>Tries to return a string representing a frequency guess, generated by infer_freq.</td>
</tr>
</tbody>
</table>

**pandas.TimedeltaIndex.days**

*property* TimedeltaIndex.days

Number of days for each element.

**pandas.TimedeltaIndex.seconds**

*property* TimedeltaIndex.seconds

Number of seconds (>= 0 and less than 1 day) for each element.

**pandas.TimedeltaIndex.microseconds**

*property* TimedeltaIndex.microseconds

Number of microseconds (>= 0 and less than 1 second) for each element.

**pandas.TimedeltaIndex.nanoseconds**

*property* TimedeltaIndex.nanoseconds

Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
**pandas.TimedeltaIndex.components**

**property TimedeltaIndex.components**

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns

DataFrame

**pandas.TimedeltaIndex.inferred_freq**

**TimedeltaIndex.inferred_freq**

Tries to return a string representing a frequency guess, generated by infer_freq. Returns None if it can’t autodetect the frequency.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_pytimedelta(*args, **kwargs)</td>
<td>Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.</td>
</tr>
<tr>
<td>to_series([index, name])</td>
<td>Create a Series with both index and values equal to the index keys.</td>
</tr>
<tr>
<td>round(*args, **kwargs)</td>
<td>Perform round operation on the data to the specified freq.</td>
</tr>
<tr>
<td>floor(*args, **kwargs)</td>
<td>Perform floor operation on the data to the specified freq.</td>
</tr>
<tr>
<td>ceil(*args, **kwargs)</td>
<td>Perform ceil operation on the data to the specified freq.</td>
</tr>
<tr>
<td>to_frame([index, name])</td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
<tr>
<td>mean(*args, **kwargs)</td>
<td>Return the mean value of the Array.</td>
</tr>
</tbody>
</table>

**pandas.TimedeltaIndex.to_pytimedelta**

**TimedeltaIndex.to_pytimedelta(*args, **kwargs)**

Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.

Returns

timedeltas [ndarray[object]]
pandas.TimedeltaIndex.to_series

TimedeltaIndex.to_series(index=None, name=None)
Create a Series with both index and values equal to the index keys.
Useful with map for returning an indexer based on an index.

Parameters

- **index** [Index, optional] Index of resulting Series. If None, defaults to original index.
- **name** [str, optional] Name of resulting Series. If None, defaults to name of original index.

Returns

Series The dtype will be based on the type of the Index values.

See also:

- Index.to_frame Convert an Index to a DataFrame.
- Series.to_frame Convert Series to DataFrame.

Examples

```python
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')

By default, the original Index and original name is reused.
```
```
>>> idx.to_series()
animal
Ant   Ant
Bear  Bear
Cow   Cow
Name: animal, dtype: object
```

To enforce a new Index, specify new labels to **index**:
```
>>> idx.to_series(index=[0, 1, 2])
0   Ant
1   Bear
2   Cow
Name: animal, dtype: object
```

To override the name of the resulting column, specify **name**:
```
>>> idx.to_series(name='zoo')
animal
Ant   Ant
Bear  Bear
Cow   Cow
Name: zoo, dtype: object
```
**pandas.TimedeltaIndex.round**

TimedeltaIndex.round(*args, **kwargs)

Perform round operation on the data to the specified `freq`.

**Parameters**

- `freq` [str or Offset] The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible `freq` values.
- `ambiguous` ['infer', bool-ndarray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:
  - ‘infer’ will attempt to infer fall dst-transition hours based on order
  - bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
  - ‘NaT’ will return NaT where there are ambiguous times
  - ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.
- `nonexistent` ['shift_forward', 'shift_backward', ‘NaT’, timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
  - ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
  - ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
  - ‘NaT’ will return NaT where there are nonexistent times
  - timedelta objects will shift nonexistent times by the timedelta
  - ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

**Returns**

- DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

- ValueError if the `freq` cannot be converted.

**Notes**

If the timestamps have a timezone, rounding will take place relative to the local (“wall”) time and re-localized to the same timezone. When rounding near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.
Examples

DatetimeIndex

>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
               dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
               dtype='datetime64[ns]', freq=None)

Series

>>> pd.Series(rng).dt.round("H")
0 2018-01-01 12:00:00
1 2018-01-01 12:00:00
2 2018-01-01 12:00:00
dtype: datetime64[ns]

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

>>> rng_tz = pd.DatetimeIndex(['2021-10-31 03:30:00'], tz="Europe/Amsterdam")
>>> rng_tz.floor("2H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)

>>> rng_tz.floor("2H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)

pandas.TimedeltaIndex.floor

TimedeltaIndex.floor(*args, **kwargs)

Perform floor operation on the data to the specified freq.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.

ambiguous [‘infer’, bool-ndarray, ‘NaT’, default ‘raise’] Only relevant for DatetimeIndex:
  • ‘infer’ will attempt to infer fall dst-transition hours based on order
  • bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
  • ‘NaT’ will return NaT where there are ambiguous times

3.6. Index objects 2277
• ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

**nonexistent** ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

• ‘shift_forward’ will shift the nonexistent time forward to the closest existing time

• ‘shift_backward’ will shift the nonexistent time backward to the closest existing time

• ‘NaT’ will return NaT where there are nonexistent times

• timedelta objects will shift nonexistent times by the timedelta

• ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns

**DatetimeIndex, TimedeltaIndex, or Series** Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

ValueError if the freq cannot be converted.

Notes

If the timestamps have a timezone, flooring will take place relative to the local (“wall”) time and re-localized to the same timezone. When flooring near daylight savings time, use **nonexistent** and **ambiguous** to control the re-localization behavior.

Examples

**DatetimeIndex**

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
```

```python
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**Series**

```python
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use **ambiguous** or **nonexistent** to control how the timestamp should be re-localized.
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 03:30:00'], tz="Europe/Amsterdam")

>>> rng_tz.floor("2H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
        dtype='datetime64[ns, Europe/Amsterdam]', freq=None)

>>> rng_tz.floor("2H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
        dtype='datetime64[ns, Europe/Amsterdam]', freq=None)

pandas.TimedeltaIndex.ceil

TimedeltaIndex.ceil(*args, **kwargs)
Perform ceil operation on the data to the specified freq.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See frequency aliases for a list of possible freq values.

ambiguous ['infer', bool-ndarray, 'NaT', default ‘raise’] Only relevant for DatetimeIndex:
- ‘infer’ will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- ‘NaT’ will return NaT where there are ambiguous times
- ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

nonexistent ['shift_forward', 'shift_backward', 'NaT', timedelta, default ‘raise’] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.
- ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
- ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
- ‘NaT’ will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- ‘raise’ will raise an NonExistentTimeError if there are nonexistent times.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the freq cannot be converted.
Notes

If the timestamps have a timezone, ceiling will take place relative to the local ("wall") time and re-localized to the same timezone. When ceiling near daylight savings time, use nonexistent and ambiguous to control the re-localization behavior.

Examples

DatetimeIndex

```python
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00', '2018-01-01 12:01:00'],
               dtype='datetime64[ns]', freq='T')
```

```python
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00', '2018-01-01 13:00:00'],
               dtype='datetime64[ns]', freq=None)
```

Series

```python
>>> pd.Series(rng).dt.ceil("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 13:00:00
dtype: datetime64[ns]
```

When rounding near a daylight savings time transition, use ambiguous or nonexistent to control how the timestamp should be re-localized.

```python
>>> rng_tz = pd.DatetimeIndex(['2021-10-31 01:30:00'], tz="Europe/Amsterdam")
```

```python
>>> rng_tz.ceil("H", ambiguous=False)
DatetimeIndex(['2021-10-31 02:00:00+01:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

```python
>>> rng_tz.ceil("H", ambiguous=True)
DatetimeIndex(['2021-10-31 02:00:00+02:00'],
               dtype='datetime64[ns, Europe/Amsterdam]', freq=None)
```

pandas.TimedeltaIndex.to_frame

TimedeltaIndex.to_frame(index=True, name=NoDefault.no_default)
Create a DataFrame with a column containing the Index.

Parameters

- **index** [bool, default True] Set the index of the returned DataFrame as the original Index.
- **name** [object, default None] The passed name should substitute for the index name (if it has one).

Returns
DataFrame DataFrame containing the original Index data.

See also:

Index.to_series Convert an Index to a Series.
Series.to_frame Convert Series to DataFrame.

Examples

```python
class pd:
    def index(self, *args, **kwargs):
        self.to_frame(*args, **kwargs)

    def to_frame(self, index=False, name='animal'):
        if index:
            return self
        else:
            return self
```

```python
>>> index = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> index.to_frame()
     animal
      Ant    Ant
    Bear    Bear
    Cow     Cow
```

By default, the original Index is reused. To enforce a new Index:

```python
>>> index.to_frame(index=False)
     animal
      0    Ant
      1    Bear
      2    Cow
```

To override the name of the resulting column, specify name:

```python
>>> index.to_frame(index=False, name='zoo')
     zoo
      0    Ant
      1    Bear
      2    Cow
```

pandas.TimedeltaIndex.mean

TimedeltaIndex.mean(*args, **kwargs)

Return the mean value of the Array.

New in version 0.25.0.

Parameters

skipna [bool, default True] Whether to ignore any NaT elements.
axis [int, optional, default 0]

Returns

scalar Timestamp or Timedelta.

See also:

numpy.ndarray.mean Returns the average of array elements along a given axis.
Series.mean Return the mean value in a Series.
Notes

mean is only defined for Datetime and Timedelta dtypes, not for Period.

Components

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimedeltaIndex.days</td>
<td>Number of days for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.seconds</td>
<td>Number of seconds (&gt;= 0 and less than 1 day) for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.microseconds</td>
<td>Number of microseconds (&gt;= 0 and less than 1 second) for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.nanoseconds</td>
<td>Number of nanoseconds (&gt;= 0 and less than 1 microsecond) for each element.</td>
</tr>
<tr>
<td>TimedeltaIndex.components</td>
<td>Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.</td>
</tr>
<tr>
<td>TimedeltaIndex.inferred_freq</td>
<td>Tries to return a string representing a frequency guess, generated by infer_freq.</td>
</tr>
</tbody>
</table>

Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimedeltaIndex.to_pytimedelta(*args, **kwargs)</td>
<td>Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.</td>
</tr>
<tr>
<td>TimedeltaIndex.to_series([index, name])</td>
<td>Create a Series with both index and values equal to the index keys.</td>
</tr>
<tr>
<td>TimedeltaIndex.round(*args, **kwargs)</td>
<td>Perform round operation on the data to the specified freq.</td>
</tr>
<tr>
<td>TimedeltaIndex.floor(*args, **kwargs)</td>
<td>Perform floor operation on the data to the specified freq.</td>
</tr>
<tr>
<td>TimedeltaIndex.ceil(*args, **kwargs)</td>
<td>Perform ceil operation on the data to the specified freq.</td>
</tr>
<tr>
<td>TimedeltaIndex.to_frame([index, name])</td>
<td>Create a DataFrame with a column containing the Index.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimedeltaIndex.mean(*args, **kwargs)</td>
<td>Return the mean value of the Array.</td>
</tr>
</tbody>
</table>

3.6.8 PeriodIndex

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PeriodIndex([data, ordinal, freq, dtype, ...])</td>
<td>Immutable ndarray holding ordinal values indicating regular periods in time.</td>
</tr>
</tbody>
</table>
pandas.PeriodIndex

class pandas.PeriodIndex(data=None, ordinal=None, freq=None, dtype=None, copy=False, name=None, **fields)

Immutable ndarray holding ordinal values indicating regular periods in time.
Index keys are boxed to Period objects which carries the metadata (eg, frequency information).

Parameters

data [array-like (1d int np.ndarray or PeriodArray), optional] Optional period-like data to construct index with.
copy [bool] Make a copy of input ndarray.
freq [str or period object, optional] One of pandas period strings or corresponding objects.
year [int, array, or Series, default None]
month [int, array, or Series, default None]
quarter [int, array, or Series, default None]
day [int, array, or Series, default None]
hour [int, array, or Series, default None]
minute [int, array, or Series, default None]
second [int, array, or Series, default None]
dtype [str or PeriodDtype, default None]

See also:
Index The base pandas Index type.
Period Represents a period of time.
DatetimeIndex Index with datetime64 data.
TimedeltaIndex Index of timedelta64 data.
period_range Create a fixed-frequency PeriodIndex.

Examples

>>> idx = pd.PeriodIndex(year=[2000, 2002], quarter=[1, 3])
>>> idx
PeriodIndex(['2000Q1', '2002Q3'], dtype='period[Q-DEC]')
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>day</code></td>
<td>The days of the period.</td>
</tr>
<tr>
<td><code>dayofweek</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>day_of_week</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>dayofyear</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>day_of_year</code></td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td><code>days_in_month</code></td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td><code>daysinmonth</code></td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td><code>freq</code></td>
<td>Return the frequency object if it is set, otherwise None.</td>
</tr>
<tr>
<td><code>freqstr</code></td>
<td>Return the frequency object as a string if its set, otherwise None.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>The hour of the period.</td>
</tr>
<tr>
<td><code>is_leap_year</code></td>
<td>Logical indicating if the date belongs to a leap year.</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>The minute of the period.</td>
</tr>
<tr>
<td><code>month</code></td>
<td>The month as January=1, December=12.</td>
</tr>
<tr>
<td><code>quarter</code></td>
<td>The quarter of the date.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>The second of the period.</td>
</tr>
<tr>
<td><code>week</code></td>
<td>The week ordinal of the year.</td>
</tr>
<tr>
<td><code>weekday</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>weekofyear</code></td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td><code>year</code></td>
<td>The year of the period.</td>
</tr>
</tbody>
</table>

**pandas.PeriodIndex.day**

**property** `PeriodIndex.day`

The days of the period.

**pandas.PeriodIndex.dayofweek**

**property** `PeriodIndex.dayofweek`

The day of the week with Monday=0, Sunday=6.

**pandas.PeriodIndex.day_of_week**

**property** `PeriodIndex.day_of_week`

The day of the week with Monday=0, Sunday=6.
pandas.PeriodIndex.dayofyear

property PeriodIndex.dayofyear
   The ordinal day of the year.

pandas.PeriodIndex.day_of_year

property PeriodIndex.day_of_year
   The ordinal day of the year.

pandas.PeriodIndex.days_in_month

property PeriodIndex.days_in_month
   The number of days in the month.

pandas.PeriodIndex.daysinmonth

property PeriodIndex.daysinmonth
   The number of days in the month.

pandas.PeriodIndex.freq

property PeriodIndex.freq
   Return the frequency object if it is set, otherwise None.

pandas.PeriodIndex.freqstr

property PeriodIndex.freqstr
   Return the frequency object as a string if its set, otherwise None.

pandas.PeriodIndex.hour

property PeriodIndex.hour
   The hour of the period.

pandas.PeriodIndex.is_leap_year

property PeriodIndex.is_leap_year
   Logical indicating if the date belongs to a leap year.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.PeriodIndex.minute

property PeriodIndex.minute
The minute of the period.

pandas.PeriodIndex.month

property PeriodIndex.month
The month as January=1, December=12.

pandas.PeriodIndex.quarter

property PeriodIndex.quarter
The quarter of the date.

pandas.PeriodIndex.second

property PeriodIndex.second
The second of the period.

pandas.PeriodIndex.week

property PeriodIndex.week
The week ordinal of the year.

pandas.PeriodIndex.weekday

property PeriodIndex.weekday
The day of the week with Monday=0, Sunday=6.

pandas.PeriodIndex.weekofyear

property PeriodIndex.weekofyear
The week ordinal of the year.

pandas.PeriodIndex.year

property PeriodIndex.year
The year of the period.
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asfreq([freq, how])</code></td>
<td>Convert the PeriodArray to the specified frequency <code>freq</code>.</td>
</tr>
<tr>
<td><code>strftime(*args, **kwargs)</code></td>
<td>Convert to Index using specified date_format.</td>
</tr>
<tr>
<td><code>to_timestamp([freq, how])</code></td>
<td>Cast to DatetimeArray/Index.</td>
</tr>
</tbody>
</table>

#### pandas.PeriodIndex.asfreq

PeriodIndex.asfreq(freq=None, how='E')

Convert the PeriodArray to the specified frequency `freq`.

Equivalent to applying `pandas.Period.asfreq()` with the given arguments to each Period in this PeriodArray.

**Parameters**

- `freq` [str] A frequency.
- `how` [str {‘E’, ‘S’}, default ‘E’] Whether the elements should be aligned to the end or start within a period.
  - ‘E’, ‘END’, or ‘FINISH’ for end,
  - ‘S’, ‘START’, or ‘BEGIN’ for start.
  January 31st (‘END’) vs. January 1st (‘START’) for example.

**Returns**

PeriodArray The transformed PeriodArray with the new frequency.

**See also:**

- `pandas.arrays.PeriodArray.asfreq` Convert each Period in a PeriodArray to the given frequency.
- `Period.asfreq` Convert a Period object to the given frequency.

#### Examples

```python
g>>> pidx = pd.period_range('2010-01-01', '2015-01-01', freq='A')
g>>> pidx

g>>> pidx.asfreq('M')
PeriodIndex(['2010-12', '2011-12', '2012-12', '2013-12', '2014-12', '2015-12'], dtype='period[M]')

g>>> pidx.asfreq('M', how='S')
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.PeriodIndex.strftime

PeriodIndex.strftime(*args, **kwargs)
Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

Parameters

date_format [str] Date format string (e.g. “%Y-%m-%d”).

Returns

ndarray[object] NumPy ndarray of formatted strings.

See also:

to_datetime Convert the given argument to datetime.

DatetimeIndex.normalize Return DatetimeIndex with times to midnight.

DatetimeIndex.round Round the DatetimeIndex to the specified freq.

DatetmeIndex.floor Floor the DatetimeIndex to the specified freq.

Examples

```python
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
                      periods=3, freq='s')
>>> rng.strftime(\n"%B \%d, \%Y, %r")
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
      'March 10, 2018, 09:00:02 AM'],
     dtype='object')
```

pandas.PeriodIndex.to_timestamp

PeriodIndex.to_timestamp(freq=None, how='start')
Cast to DatetimeArray/Index.

Parameters

freq [str or DateOffset, optional] Target frequency. The default is ‘D’ for week or longer, ‘S’ otherwise.

how [‘s’, ‘e’, ‘start’, ‘end’] Whether to use the start or end of the time period being converted.

Returns

DatetimeArray/Index
## Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PeriodIndex.day</td>
<td>The days of the period.</td>
</tr>
<tr>
<td>PeriodIndex.dayofweek</td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td>PeriodIndex.day_of_week</td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td>PeriodIndex.dayofyear</td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td>PeriodIndex.day_of_year</td>
<td>The ordinal day of the year.</td>
</tr>
<tr>
<td>PeriodIndex.days_in_month</td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td>PeriodIndex.daysinmonth</td>
<td>The number of days in the month.</td>
</tr>
<tr>
<td>PeriodIndex.end_time</td>
<td>The hour of the period.</td>
</tr>
<tr>
<td>PeriodIndex.freq</td>
<td>Return the frequency object if it is set, otherwise None.</td>
</tr>
<tr>
<td>PeriodIndex.freqstr</td>
<td>Return the frequency object as a string if its set, otherwise None.</td>
</tr>
<tr>
<td>PeriodIndex.is_leap_year</td>
<td>Logical indicating if the date belongs to a leap year.</td>
</tr>
<tr>
<td>PeriodIndex.minute</td>
<td>The minute of the period.</td>
</tr>
<tr>
<td>PeriodIndex.month</td>
<td>The month as January=1, December=12.</td>
</tr>
<tr>
<td>PeriodIndex.quarter</td>
<td>The quarter of the date.</td>
</tr>
<tr>
<td>PeriodIndex.qyear</td>
<td>The second of the period.</td>
</tr>
<tr>
<td>PeriodIndex.second</td>
<td>The year of the period.</td>
</tr>
<tr>
<td>PeriodIndex.start_time</td>
<td>The week ordinal of the year.</td>
</tr>
<tr>
<td>PeriodIndex.weekday</td>
<td>The day of the week with Monday=0, Sunday=6.</td>
</tr>
<tr>
<td>PeriodIndex.weekofyear</td>
<td>The week ordinal of the year.</td>
</tr>
<tr>
<td>PeriodIndex.year</td>
<td>The year of the period.</td>
</tr>
</tbody>
</table>

### pandas.PeriodIndex.end_time

**property** PeriodIndex.end_time

### pandas.PeriodIndex.qyear

**property** PeriodIndex.qyear

### pandas.PeriodIndex.start_time

**property** PeriodIndex.start_time
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`PeriodIndex.asfreq(freq, how)</td>
<td>Convert the PeriodArray to the specified frequency <code>freq</code>.</td>
</tr>
<tr>
<td>`PeriodIndex.strftime(*args, **kwargs)</td>
<td>Convert to Index using specified date_format.</td>
</tr>
<tr>
<td>`PeriodIndex.to_timestamp(freq, how)</td>
<td>Cast to DatetimeArray/Index.</td>
</tr>
</tbody>
</table>

3.7 Date offsets

3.7.1 DateOffset

```python
class pandas.tseries.offsets.DateOffset
```

Standard kind of date increment used for a date range.

Works exactly like the keyword argument form of relativedelta. Note that the positional argument form of relativedelta is not supported. Use of the keyword n is discouraged—you would be better off specifying n in the keywords you use, but regardless it is there for you. n is needed for DateOffset subclasses.

DateOffset works as follows. Each offset specify a set of dates that conform to the DateOffset. For example, Bday defines this set to be the set of dates that are weekdays (M-F). To test if a date is in the set of a DateOffset dateOffset we can use the is_on_offset method: dateOffset.is_on_offset(date).

If a date is not on a valid date, the rollback and rollforward methods can be used to roll the date to the nearest valid date before/after the date.

DateOffsets can be created to move dates forward a given number of valid dates. For example, Bday(2) can be added to a date to move it two business days forward. If the date does not start on a valid date, first it is moved to a valid date. Thus pseudo code is:

```python
def __add__(date): date = rollback(date) # does nothing if date is valid return date + <n number of periods>
```

When a date offset is created for a negative number of periods, the date is first rolled forward. The pseudo code is:

```python
def __add__(date): date = rollforward(date) # does nothing is date is valid return date + <n number of periods>
```

Zero presents a problem. Should it roll forward or back? We arbitrarily have it rollforward:

```
date + BDay(0) == BDay.rollforward(date)
```

Since 0 is a bit weird, we suggest avoiding its use.

**Parameters**

- `n` [int, default 1] The number of time periods the offset represents. If specified without a temporal pattern, defaults to n days.
- `normalize` [bool, default False] Whether to round the result of a DateOffset addition down to the previous midnight.
- `**kwds` Temporal parameter that add to or replace the offset value.

Parameters that add to the offset (like Timedelta):

- years
- months
• weeks
• days
• hours
• minutes
• seconds
• microseconds
• nanoseconds

Parameters that replace the offset value:
• year
• month
• day
• weekday
• hour
• minute
• second
• microsecond
• nanosecond.

See also:
dateutil.relativedelta.relativedelta The relativedelta type is designed to be applied to an existing datetime and can replace specific components of that datetime, or represents an interval of time.

Examples

```python
>>> from pandas.tseries.offsets import DateOffset
>>> ts = pd.Timestamp('2017-01-01 09:10:11')
>>> ts + DateOffset(months=3)
Timestamp('2017-04-01 09:10:11')

>>> ts = pd.Timestamp('2017-01-01 09:10:11')
>>> ts + DateOffset(months=2)
Timestamp('2017-03-01 09:10:11')
```

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

3.7. Date offsets
pandas.tseries.offsets.DateOffset.base

DateOffset.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th>kwds</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
</tr>
</thead>
</table>

Methods

___call__(*args, **kwargs)

Call self as a function.

rollback

Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.DateOffset.rollback

DateOffset.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

rollforward

Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
3.7. Date offsets

pandas: powerful Python data analysis toolkit, Release 1.4.2

Properties

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateOffset.freqstr</td>
</tr>
<tr>
<td>DateOffset.kwds</td>
</tr>
<tr>
<td>DateOffset.name</td>
</tr>
<tr>
<td>DateOffset.nanos</td>
</tr>
<tr>
<td>DateOffset.normalize</td>
</tr>
<tr>
<td>DateOffset.rule_code</td>
</tr>
<tr>
<td>DateOffset.n</td>
</tr>
<tr>
<td>DateOffset.is_month_start</td>
</tr>
<tr>
<td>DateOffset.is_month_end</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.DateOffset.freqstr

DateOffset.freqstr
pandas: powerful Python data analysis toolkit, Release 1.4.2

`pandas.tseries.offsets.DateOffset.kwds`

DateOffset.kwds

`pandas.tseries.offsets.DateOffset.name`

DateOffset.name

`pandas.tseries.offsets.DateOffset.nanos`

DateOffset.nanos

`pandas.tseries.offsets.DateOffset.normalize`

DateOffset.normalize

`pandas.tseries.offsets.DateOffset.rule_code`

DateOffset.rule_code

`pandas.tseries.offsets.DateOffset.n`

DateOffset.n

`pandas.tseries.offsets.DateOffset.is_month_start`

DateOffset.is_month_start()

`pandas.tseries.offsets.DateOffset.is_month_end`

DateOffset.is_month_end()
Methods

- `DateOffset.apply`
- `DateOffset.apply_index(other)`
- `DateOffset.copy`
- `DateOffset.isAnchored`
- `DateOffset.onOffset`
- `DateOffset.is_anchored`
- `DateOffset.is_on_offset`
- `DateOffset.__call__(*args,**kwargs)` Call self as a function.
- `DateOffset.is_month_start`
- `DateOffset.is_month_end`
- `DateOffset.is_quarter_start`
- `DateOffset.is_quarter_end`
- `DateOffset.is_year_start`
- `DateOffset.is_year_end`

**pandas.tseries.offsets.DateOffset.apply**

`DateOffset.apply()`

**pandas.tseries.offsets.DateOffset.apply_index**

`DateOffset.apply_index(other)`

**pandas.tseries.offsets.DateOffset.copy**

`DateOffset.copy()`
pandas.tseries.offsets.DateOffset.isAnchored

DateOffset.isAnchored()

pandas.tseries.offsets.DateOffset.onOffset

DateOffset.onOffset()

pandas.tseries.offsets.DateOffset.is_anchored

DateOffset.is_anchored()

pandas.tseries.offsets.DateOffset.is_on_offset

DateOffset.is_on_offset()

pandas.tseries.offsets.DateOffset.is_quarter_start

DateOffset.is_quarter_start()

pandas.tseries.offsets.DateOffset.is_quarter_end

DateOffset.is_quarter_end()

pandas.tseries.offsets.DateOffset.is_year_start

DateOffset.is_year_start()

pandas.tseries.offsets.DateOffset.is_year_end

DateOffset.is_year_end()

### 3.7.2 BusinessDay

| BusinessDay | DateOffset subclass representing possibly n business days. |
pandas.tseries.offsets.BusinessDay

class pandas.tseries.offsets.BusinessDay

DateOffset subclass representing possibly n business days.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Returns a copy of the calling offset object with n=1 and all other attributes equal.</td>
</tr>
<tr>
<td>offset</td>
<td>Alias for self._offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.BusinessDay.base

BusinessDay.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BusinessDay.offset

BusinessDay.offset

Alias for self._offset.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessDay.__call__

BusinessDay.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.BusinessDay.rollback

BusinessDay.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.BusinessDay.rollforward

BusinessDay.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>apply_index</td>
<td></td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
</tr>
</tbody>
</table>

Alias:

BDay alias of pandas._libs.tslibs.offsets.BusinessDay
pandas.tseries.offsets.BDay

alias of pandas._libs.tslibs.offsets.BusinessDay

Properties

- BusinessDay.freqstr
- BusinessDay.kwds
- BusinessDay.name
- BusinessDay.nanos
- BusinessDay.normalize
- BusinessDay.rule_code
- BusinessDay.n
- BusinessDay.weekmask
- BusinessDay.holidays
- BusinessDay.calendar

pandas.tseries.offsets.BusinessDay.freqstr

BusinessDay.freqstr

pandas.tseries.offsets.BusinessDay.kwds

BusinessDay.kwds

pandas.tseries.offsets.BusinessDay.name

BusinessDay.name
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessDay.nanos

BusinessDay.nanos

pandas.tseries.offsets.BusinessDay.normalize

BusinessDay.normalize

pandas.tseries.offsets.BusinessDay.rule_code

BusinessDay.rule_code

pandas.tseries.offsets.BusinessDay.n

BusinessDay.n

pandas.tseries.offsets.BusinessDay.weekmask

BusinessDay.weekmask

pandas.tseries.offsets.BusinessDay.holidays

BusinessDay.holidays

pandas.tseries.offsets.BusinessDay.calendar

BusinessDay.calendar
Methods

Methods available in pandas.tseries.offsets.BusinessDay:

- BusinessDay.apply
- BusinessDay.apply_index(other)
- BusinessDay.copy
- BusinessDay.isAnchored
- BusinessDay.onOffset
- BusinessDay.is_anchored
- BusinessDay.is_on_offset
- BusinessDay.__call__(*args, **kwargs) Call self as a function.
- BusinessDay.is_month_start
- BusinessDay.is_month_end
- BusinessDay.is_quarter_start
- BusinessDay.is_quarter_end
- BusinessDay.is_year_start
- BusinessDay.is_year_end

pandas.tseries.offsets.BusinessDay.apply

BusinessDay.apply()

pandas.tseries.offsets.BusinessDay.apply_index

BusinessDay.apply_index(other)

pandas.tseries.offsets.BusinessDay.copy

BusinessDay.copy()
pandas.tseries.offsets.BusinessDay.isAnchored

BusinessDay.isAnchored()

pandas.tseries.offsets.BusinessDay.onOffset

BusinessDay.onOffset()

pandas.tseries.offsets.BusinessDay.is_anchored

BusinessDay.is_anchored()

pandas.tseries.offsets.BusinessDay.is_on_offset

BusinessDay.is_on_offset()

pandas.tseries.offsets.BusinessDay.is_month_start

BusinessDay.is_month_start()

pandas.tseries.offsets.BusinessDay.is_month_end

BusinessDay.is_month_end()

pandas.tseries.offsets.BusinessDay.is_quarter_start

BusinessDay.is_quarter_start()

pandas.tseries.offsets.BusinessDay.is_quarter_end

BusinessDay.is_quarter_end()

pandas.tseries.offsets.BusinessDay.is_year_start

BusinessDay.is_year_start()
pandas.tseries.offsets.BusinessDay.is_year_end

BusinessDay.is_year_end()

### 3.7.3 BusinessHour

<table>
<thead>
<tr>
<th>BusinessHour</th>
<th>DateOffset subclass representing possibly n business hours.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.BusinessHour**

**class pandas.tseries.offsets.BusinessHour**

DateOffset subclass representing possibly n business hours.

**Parameters**

- **n** [int, default 1] The number of months represented.
- **normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.
- **weekmask** [str, Default ‘Mon Tue Wed Thu Fri’] Weekmask of valid business days, passed to numpy.busdaycalendar.
- **start** [str, default “09:00”] Start time of your custom business hour in 24h format.
- **end** [str, default: “17:00”] End time of your custom business hour in 24h format.

**Attributes**

- **base** Returns a copy of the calling offset object with n=1 and all other attributes equal.
- **next_bday** Used for moving to next business day.
- **offset** Alias for self._offset.

**pandas.tseries.offsets.BusinessHour.base**

BusinessHour.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

**pandas.tseries.offsets.BusinessHour.next_bday**

BusinessHour.next_bday

Used for moving to next business day.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessHour.offset

BusinessHour.offset
Alias for self._offset.

calendar
end
freqstr
holidays
kwds
n
name
nanos
normalize
rule_code
start
weekmask

Methods

__call__(*args, **kwargs) Call self as a function.

rollback(other) Roll provided date backward to next offset only if not on offset.

rollforward(other) Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessHour.__call__

BusinessHour.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.BusinessHour.rollback

BusinessHour.rollback(other)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BusinessHour.rollforward

BusinessHour.rollforward(other)
Roll provided date forward to next offset only if not on offset.
3.7. Date offsets

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
</tr>
</tbody>
</table>

**Properties**

- `BusinessHour.freqstr`
- `BusinessHour.kwds`
- `BusinessHour.name`
- `BusinessHour.nanos`
- `BusinessHour.normalize`
- `BusinessHour.rule_code`
- `BusinessHour.n`
- `BusinessHour.start`
- `BusinessHour.end`
- `BusinessHour.weekmask`
- `BusinessHour.holidays`
- `BusinessHour.calendar`
pandas: powerful Python data analysis toolkit, Release 1.4.2

- `pandas.tseries.offsets.BusinessHour.freqstr`
  ```
  BusinessHour.freqstr
  ```

- `pandas.tseries.offsets.BusinessHour.kwds`
  ```
  BusinessHour.kwds
  ```

- `pandas.tseries.offsets.BusinessHour.name`
  ```
  BusinessHour.name
  ```

- `pandas.tseries.offsets.BusinessHour.nanos`
  ```
  BusinessHour.nanos
  ```

- `pandas.tseries.offsets.BusinessHour.normalize`
  ```
  BusinessHour.normalize
  ```

- `pandas.tseries.offsets.BusinessHour.rule_code`
  ```
  BusinessHour.rule_code
  ```

- `pandas.tseries.offsets.BusinessHour.n`
  ```
  BusinessHour.n
  ```

- `pandas.tseries.offsets.BusinessHour.start`
  ```
  BusinessHour.start
  ```

- `pandas.tseries.offsets.BusinessHour.end`
  ```
  BusinessHour.end
  ```
pandas.tseries.offsets.BusinessHour.weekmask

BusinessHour.weekmask

pandas.tseries.offsets.BusinessHour.holidays

BusinessHour.holidays

pandas.tseries.offsets.BusinessHour.calendar

BusinessHour.calendar

Methods

BusinessHour.apply

BusinessHour.apply_index(other)

BusinessHour.copy

BusinessHour.isAnchored

BusinessHour.onOffset

BusinessHour.is_anchored

BusinessHour.is_on_offset

BusinessHour.__call__(*args, **kwargs) Call self as a function.

BusinessHour.is_month_start

BusinessHour.is_month_end

BusinessHour.is_quarter_start

BusinessHour.is_quarter_end

BusinessHour.is_year_start

BusinessHour.is_year_end
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessHour.apply
BusinessHour.apply()

pandas.tseries.offsets.BusinessHour.apply_index
BusinessHour.apply_index(other)

pandas.tseries.offsets.BusinessHour.copy
BusinessHour.copy()

pandas.tseries.offsets.BusinessHour.isAnchored
BusinessHour.isAnchored()

pandas.tseries.offsets.BusinessHour.onOffset
BusinessHour.onOffset()

pandas.tseries.offsets.BusinessHour.is_anchored
BusinessHour.is_anchored()

pandas.tseries.offsets.BusinessHour.is_on_offset
BusinessHour.is_on_offset()

pandas.tseries.offsets.BusinessHour.is_month_start
BusinessHour.is_month_start()

pandas.tseries.offsets.BusinessHour.is_month_end
BusinessHour.is_month_end()
pandas.tseries.offsets.BusinessHour.is_quarter_start

BusinessHour.is_quarter_start()

pandas.tseries.offsets.BusinessHour.is_quarter_end

BusinessHour.is_quarter_end()

pandas.tseries.offsets.BusinessHour.is_year_start

BusinessHour.is_year_start()

pandas.tseries.offsets.BusinessHour.is_year_end

BusinessHour.is_year_end()

### 3.7.4 CustomBusinessDay

<table>
<thead>
<tr>
<th>CustomBusinessDay</th>
<th>DateOffset subclass representing custom business days excluding holidays.</th>
</tr>
</thead>
</table>

**class** pandas.tseries.offsets.CustomBusinessDay

DateOffset subclass representing custom business days excluding holidays.

**Parameters**

- n [int, default 1]
- normalize [bool, default False] Normalize start/end dates to midnight before generating date range.
- weekmask [str, Default ‘Mon Tue Wed Thu Fri’] Weekmask of valid business days, passed to numpy.busdaycalendar.
- holidays [list] List/array of dates to exclude from the set of valid business days, passed to numpy.busdaycalendar.
- calendar [pd.HolidayCalendar or np.busdaycalendar]
- offset [timedelta, default timedelta(0)]
### Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>Alias for self._offset.</td>
</tr>
</tbody>
</table>

#### pandas.tseries.offsets.CustomBusinessDay.base

CustomBusinessDay.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

#### pandas.tseries.offsets.CustomBusinessDay.offset

CustomBusinessDay.offset

Alias for self._offset.

<table>
<thead>
<tr>
<th>calendar</th>
<th>freqstr</th>
<th>holidays</th>
<th>kwds</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
<th>weekmask</th>
</tr>
</thead>
</table>

### Methods

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, **kwargs)</th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

#### pandas.tseries.offsets.CustomBusinessDay.__call__

CustomBusinessDay.__call__(*args, **kwargs*)

Call self as a function.
pandas.tseries.offsets.CustomBusinessDay.rollback

CustomBusinessDay.rollback()
Roll provided date backward to next offset only if not on offset.

Returns
TimeSpan  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.CustomBusinessDay.rollforward

CustomBusinessDay.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns
TimeSpan  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td>is_anchored</td>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
<td>is_on_offset</td>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
<td>is_year_end</td>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alias:

CDay  alias of pandas._libs.tslibs.offsets.CustomBusinessDay

pandas.tseries.offsets.CDay

pandas.tseries.offsets.CDay  alias of pandas._libs.tslibs.offsets.CustomBusinessDay
Properties

CustomBusinessDay.freqstr

CustomBusinessDay.kwds

CustomBusinessDay.name

CustomBusinessDay.nanos

CustomBusinessDay.normalize

CustomBusinessDay.rule_code

CustomBusinessDay.n

CustomBusinessDay.weekmask

CustomBusinessDay.calendar

CustomBusinessDay.holidays

pandas.tseries.offsets.CustomBusinessDay.freqstr

CustomBusinessDay.freqstr

pandas.tseries.offsets.CustomBusinessDay.kwds

CustomBusinessDay.kwds

pandas.tseries.offsets.CustomBusinessDay.name

CustomBusinessDay.name

pandas.tseries.offsets.CustomBusinessDay.nanos

CustomBusinessDay.nanos
pandas.tseries.offsets.CustomBusinessDay.normalize

CustomBusinessDay.normalize

pandas.tseries.offsets.CustomBusinessDay.rule_code

CustomBusinessDay.rule_code

pandas.tseries.offsets.CustomBusinessDay.n

CustomBusinessDay.n

pandas.tseries.offsets.CustomBusinessDay.weekmask

CustomBusinessDay.weekmask

pandas.tseries.offsets.CustomBusinessDay.calendar

CustomBusinessDay.calendar

pandas.tseries.offsets.CustomBusinessDay.holidays

CustomBusinessDay.holidays

3.7. Date offsets
Methods

CustomBusinessDay.apply_index

CustomBusinessDay.apply

CustomBusinessDay.copy

CustomBusinessDay.isAnchored

CustomBusinessDay.onOffset

CustomBusinessDay.is_anchored

CustomBusinessDay.is_on_offset

CustomBusinessDay._call_(*args, **kwargs)  Call self as a function.

CustomBusinessDay.is_month_start

CustomBusinessDay.is_month_end

CustomBusinessDay.is_quarter_start

CustomBusinessDay.is_quarter_end

CustomBusinessDay.is_year_start

CustomBusinessDay.is_year_end

pandas.tseries.offsets.CustomBusinessDay.apply_index

CustomBusinessDay.apply_index()

pandas.tseries.offsets.CustomBusinessDay.apply

CustomBusinessDay.apply()

pandas.tseries.offsets.CustomBusinessDay.copy

CustomBusinessDay.copy()
pandas.tseries.offsets.CustomBusinessDay.isAnchored

CustomBusinessDay.isAnchored()

pandas.tseries.offsets.CustomBusinessDay.onOffset

CustomBusinessDay.onOffset()

pandas.tseries.offsets.CustomBusinessDay.is_anchored

CustomBusinessDay.is_anchored()

pandas.tseries.offsets.CustomBusinessDay.is_on_offset

CustomBusinessDay.is_on_offset()

pandas.tseries.offsets.CustomBusinessDay.is_month_start

CustomBusinessDay.is_month_start()

pandas.tseries.offsets.CustomBusinessDay.is_month_end

CustomBusinessDay.is_month_end()

pandas.tseries.offsets.CustomBusinessDay.is_quarter_start

CustomBusinessDay.is_quarter_start()

pandas.tseries.offsets.CustomBusinessDay.is_quarter_end

CustomBusinessDay.is_quarter_end()

pandas.tseries.offsets.CustomBusinessDay.is_year_start

CustomBusinessDay.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.CustomBusinessDay.is_year_end

CustomBusinessDay.is_year_end()

### 3.7.5 CustomBusinessHour

| CustomBusinessHour | DateOffset subclass representing possibly n custom business days. |

#### pandas.tseries.offsets.CustomBusinessHour

**class** pandas.tseries.offsets.CustomBusinessHour

DateOffset subclass representing possibly n custom business days.

**Parameters**

- **n** [int, default 1] The number of months represented.
- **normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.
- **weekmask** [str, Default ‘Mon Tue Wed Thu Fri’] Weekmask of valid business days, passed to numpy.busdaycalendar.
- **start** [str, default “09:00”] Start time of your custom business hour in 24h format.
- **end** [str, default: “17:00”] End time of your custom business hour in 24h format.

**Attributes**

| base | Returns a copy of the calling offset object with n=1 and all other attributes equal. |
| next_bday | Used for moving to next business day. |
| offset | Alias for self._offset. |

#### pandas.tseries.offsets.CustomBusinessHour.base

CustomBusinessHour.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

#### pandas.tseries.offsets.CustomBusinessHour.next_bday

CustomBusinessHour.next_bday

Used for moving to next business day.
CustomBusinessHour.offset

Alias for self._offset.

calendar
end
freqstr
holidays
kwds
n
name
nanos
normalize
rule_code
start
weekmask

Methods

__call__(*args, **kwargs) Call self as a function.
rollback(other) Roll provided date backward to next offset only if not on offset.
rollforward(other) Roll provided date forward to next offset only if not on offset.

CustomBusinessHour.__call__(*args, **kwargs)

Call self as a function.

CustomBusinessHour.rollback(other)

Roll provided date backward to next offset only if not on offset.

CustomBusinessHour.rollforward(other)

Roll provided date forward to next offset only if not on offset.
### CustomBusinessHour

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
</tr>
<tr>
<td>apply_index</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>isAnchored</td>
</tr>
<tr>
<td>is_anchored</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
</tr>
<tr>
<td>is_on_offset</td>
</tr>
<tr>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
</tr>
<tr>
<td>is_year_end</td>
</tr>
<tr>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
</tr>
</tbody>
</table>

**Properties**

- `CustomBusinessHour.freqstr`
- `CustomBusinessHour.kwds`
- `CustomBusinessHour.name`
- `CustomBusinessHour.nanos`
- `CustomBusinessHour.normalize`
- `CustomBusinessHour.rule_code`
- `CustomBusinessHour.n`
- `CustomBusinessHour.weekmask`
- `CustomBusinessHour.calendar`
- `CustomBusinessHour.holidays`
- `CustomBusinessHour.start`
- `CustomBusinessHour.end`
pandas.tseries.offsets.CustomBusinessHour.freqstr
CustomBusinessHour.freqstr

pandas.tseries.offsets.CustomBusinessHour.kwds
CustomBusinessHour.kwds

pandas.tseries.offsets.CustomBusinessHour.name
CustomBusinessHour.name

pandas.tseries.offsets.CustomBusinessHour.nanos
CustomBusinessHour.nanos

pandas.tseries.offsets.CustomBusinessHour.normalize
CustomBusinessHour.normalize

pandas.tseries.offsets.CustomBusinessHour.rule_code
CustomBusinessHour.rule_code

pandas.tseries.offsets.CustomBusinessHour.n
CustomBusinessHour.n

pandas.tseries.offsets.CustomBusinessHour.weekmask
CustomBusinessHour.weekmask

pandas.tseries.offsets.CustomBusinessHour.calendar
CustomBusinessHour.calendar
CustomBusinessHour.holidays

CustomBusinessHour.start

CustomBusinessHour.end

Methods

CustomBusinessHour.apply

CustomBusinessHour.apply_index(\(\text{other}\))

CustomBusinessHour.copy

CustomBusinessHour.isAnchored

CustomBusinessHour.onOffset

CustomBusinessHour.is_anchored

CustomBusinessHour.is_on_offset

CustomBusinessHour.__call__(\(*\text{args}, \**\text{kwargs}\)) Call self as a function.

CustomBusinessHour.is_month_start

CustomBusinessHour.is_month_end

CustomBusinessHour.is_quarter_start

CustomBusinessHour.is_quarter_end

CustomBusinessHour.is_year_start

CustomBusinessHour.is_year_end
pandas.tseries.offsets.CustomBusinessHour.apply

CustomBusinessHour.apply()

pandas.tseries.offsets.CustomBusinessHour.apply_index

CustomBusinessHour.apply_index(other)

pandas.tseries.offsets.CustomBusinessHour.copy

CustomBusinessHour.copy()

pandas.tseries.offsets.CustomBusinessHour.isAnchored

CustomBusinessHour.isAnchored()

pandas.tseries.offsets.CustomBusinessHour.onOffset

CustomBusinessHour.onOffset()

pandas.tseries.offsets.CustomBusinessHour.is_anchored

CustomBusinessHour.is_anchored()

pandas.tseries.offsets.CustomBusinessHour.is_on_offset

CustomBusinessHour.is_on_offset()

pandas.tseries.offsets.CustomBusinessHour.is_month_start

CustomBusinessHour.is_month_start()

pandas.tseries.offsets.CustomBusinessHour.is_month_end

CustomBusinessHour.is_month_end()
3.7.6 MonthEnd

<table>
<thead>
<tr>
<th>MonthEnd</th>
<th>DateOffset of one month end.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.MonthEnd

class pandas.tseries.offsets.MonthEnd
    DateOffset of one month end.

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.MonthEnd.base

MonthEnd.base
    Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>nanos</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>rule_code</td>
</tr>
</tbody>
</table>
Methods

```python
__call__(*args, **kwargs) Call self as a function.
rollback Roll provided date backward to next offset only if not on offset.
rollforward Roll provided date forward to next offset only if not on offset.
```

**pandas.tseries.offsets.MonthEnd.__call__**

MonthEnd.__call__(*args, **kwargs)*
Call self as a function.

**pandas.tseries.offsets.MonthEnd.rollback**

MonthEnd.rollback()
Roll provided date backward to next offset only if not on offset.

**Returns**

**TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

**pandas.tseries.offsets.MonthEnd.rollforward**

MonthEnd.rollforward()
Roll provided date forward to next offset only if not on offset.

**Returns**

**TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td>is_anchored</td>
<td>is_month_end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_month_start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_on_offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_quarter_end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_quarter_start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_year_end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is_year_start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>onOffset</td>
</tr>
</tbody>
</table>
Properties

MonthEnd.freqstr

MonthEnd.kwds

MonthEnd.name

MonthEnd.nanos

MonthEnd.normalize

MonthEnd.rule_code

MonthEnd.n

pandas.tseries.offsets.MonthEnd.freqstr

MonthEnd.freqstr

pandas.tseries.offsets.MonthEnd.kwds

MonthEnd.kwds

pandas.tseries.offsets.MonthEnd.name

MonthEnd.name

pandas.tseries.offsets.MonthEnd.nanos

MonthEnd.nanos

pandas.tseries.offsets.MonthEnd.normalize

MonthEnd.normalize
pandas.tseries.offsets.MonthEnd.rule_code

MonthEnd.rule_code

pandas.tseries.offsets.MonthEnd.n

MonthEnd.n

Methods

MonthEnd.apply

MonthEnd.apply_index(other)

MonthEnd.copy

MonthEnd.isAnchored

MonthEnd.onOffset

MonthEnd.is_anchored

MonthEnd.is_on_offset

MonthEnd.__call__(*args, **kwargs) Call self as a function.

MonthEnd.is_month_start

MonthEnd.is_month_end

MonthEnd.is_quarter_start

MonthEnd.is_quarter_end

MonthEnd.is_year_start

MonthEnd.is_year_end

pandas.tseries.offsets.MonthEnd.apply

MonthEnd.apply()
pandas.tseries.offsets.MonthEnd.apply_index

MonthEnd.apply_index(other)

pandas.tseries.offsets.MonthEnd.copy

MonthEnd.copy()

pandas.tseries.offsets.MonthEnd.isAnchored

MonthEnd.isAnchored()

pandas.tseries.offsets.MonthEnd.onOffset

MonthEnd.onOffset()

pandas.tseries.offsets.MonthEnd.is_anchored

MonthEnd.is_anchored()

pandas.tseries.offsets.MonthEnd.is_on_offset

MonthEnd.is_on_offset()

pandas.tseries.offsets.MonthEnd.is_month_start

MonthEnd.is_month_start()

pandas.tseries.offsets.MonthEnd.is_month_end

MonthEnd.is_month_end()

pandas.tseries.offsets.MonthEnd.is_quarter_start

MonthEnd.is_quarter_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

3.7.7 MonthBegin

| MonthBegin | DateOffset of one month at beginning. |

pandas.tseries.offsets.MonthBegin

class pandas.tseries.offsets.MonthBegin

    DateOffset of one month at beginning.

Attributes

- **base**
  
  Returns a copy of the calling offset object with \( n=1 \) and all other attributes equal.

pandas.tseries.offsets.MonthBegin.base

    MonthBegin.base

    Returns a copy of the calling offset object with \( n=1 \) and all other attributes equal.

    |
    | freqstr
    | kwds
    | n
    | name
    | nanos
    | normalize
    | rule_code
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__call__(*args, **kwargs)</code></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td><code>rollback</code></td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td><code>rollforward</code></td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.MonthBegin.__call__**

`MonthBegin.__call__(*args, **kwargs)`

Call self as a function.

**pandas.tseries.offsets.MonthBegin.rollback**

`MonthBegin.rollback()`

Roll provided date backward to next offset only if not on offset.

**Returns**

- **TimeStamp**  Rolled timestamp if not on offset, otherwise unchanged timestamp.

**pandas.tseries.offsets.MonthBegin.rollforward**

`MonthBegin.rollforward()`

Roll provided date forward to next offset only if not on offset.

**Returns**

- **TimeStamp**  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>apply_index</td>
<td></td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>on_offset</td>
<td></td>
</tr>
</tbody>
</table>
Properties

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MonthBegin.freqstr</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.kwds</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.name</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.nanos</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.normalize</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.rule_code</strong></td>
</tr>
<tr>
<td><strong>MonthBegin.n</strong></td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.MonthBegin.freqstr

MonthBegin.freqstr

pandas.tseries.offsets.MonthBegin.kwds

MonthBegin.kwds

pandas.tseries.offsets.MonthBegin.name

MonthBegin.name

pandas.tseries.offsets.MonthBegin.nanos

MonthBegin.nanos

pandas.tseries.offsets.MonthBegin.normalize

MonthBegin.normalize
pandas.tseries.offsets.MonthBegin.rule_code

MonthBegin.rule_code

pandas.tseries.offsets.MonthBegin.n

MonthBegin.n

Methods

MonthBegin.apply

MonthBegin.apply_index(other)

MonthBegin.copy

MonthBegin.isAnchored

MonthBegin.onOffset

MonthBegin.is_anchored

MonthBegin.is_on_offset

MonthBegin.__call__(*args, **kwargs) Call self as a function.

MonthBegin.is_month_start

MonthBegin.is_month_end

MonthBegin.is_quarter_start

MonthBegin.is_quarter_end

MonthBegin.is_year_start

MonthBegin.is_year_end

pandas.tseries.offsets.MonthBegin.apply

MonthBegin.apply()
pandas.tseries.offsets.MonthBegin.apply_index

MonthBegin.apply_index(other)

pandas.tseries.offsets.MonthBegin.copy

MonthBegin.copy()

pandas.tseries.offsets.MonthBegin.isAnchored

MonthBegin.isAnchored()

pandas.tseries.offsets.MonthBegin.onOffset

MonthBegin.onOffset()

pandas.tseries.offsets.MonthBegin.is_anchored

MonthBegin.is_anchored()

pandas.tseries.offsets.MonthBegin.is_on_offset

MonthBegin.is_on_offset()

pandas.tseries.offsets.MonthBegin.is_month_start

MonthBegin.is_month_start()

pandas.tseries.offsets.MonthBegin.is_month_end

MonthBegin.is_month_end()

pandas.tseries.offsets.MonthBegin.is_quarter_start

MonthBegin.is_quarter_start()
3.7.8 BusinessMonthEnd

<table>
<thead>
<tr>
<th>BusinessMonthEnd</th>
<th>DateOffset increments between the last business day of the month.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BusinessMonthEnd

class pandas.tseries.offsets.BusinessMonthEnd

DateOffset increments between the last business day of the month.

Examples

```python
>>> from pandas.tseries.offsets import BMonthEnd
>>> ts = pd.Timestamp('2020-05-24 05:01:15')
>>> ts + BMonthEnd()
Timestamp('2020-05-29 05:01:15')
>>> ts + BMonthEnd(2)
Timestamp('2020-06-30 05:01:15')
>>> ts + BMonthEnd(-2)
Timestamp('2020-03-31 05:01:15')
```

Attributes

| base | Returns a copy of the calling offset object with n=1 and all other attributes equal. |
pandas.tseries.offsets.BusinessMonthEnd.base

BusinessMonthEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th>kwd</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
</tr>
</thead>
</table>

Methods

__call__(*args, **kwargs)

Call self as a function.

rollback

Roll provided date backward to next offset only if not on offset.

rollforward

Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessMonthEnd.__call__

BusinessMonthEnd.__call__(*args, **kwargs)

Call self as a function.

pandas.tseries.offsets.BusinessMonthEnd.rollback

BusinessMonthEnd.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.BusinessMonthEnd.rollforward

BusinessMonthEnd.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
### pandas.tseries.offsets.BMonthEnd

```
pandas.tseries.offsets.BMonthEnd
```

Alias of `pandas._libs.tslibs.offsets.BusinessMonthEnd`

### Properties

- `BusinessMonthEnd.freqstr`
- `BusinessMonthEnd.kwds`
- `BusinessMonthEnd.name`
- `BusinessMonthEnd.nanos`
- `BusinessMonthEnd.normalize`
- `BusinessMonthEnd.rule_code`
- `BusinessMonthEnd.n`
pandas.tseries.offsets.BusinessMonthEnd.freqstr

BusinessMonthEnd.freqstr

pandas.tseries.offsets.BusinessMonthEnd.kwds

BusinessMonthEnd.kwds

pandas.tseries.offsets.BusinessMonthEnd.name

BusinessMonthEnd.name

pandas.tseries.offsets.BusinessMonthEnd.nanos

BusinessMonthEnd.nanos

pandas.tseries.offsets.BusinessMonthEnd.normalize

BusinessMonthEnd.normalize

pandas.tseries.offsets.BusinessMonthEnd.rule_code

BusinessMonthEnd.rule_code

pandas.tseries.offsets.BusinessMonthEnd.n

BusinessMonthEnd.n
Methods

`BusinessMonthEnd.apply`

`BusinessMonthEnd.apply_index(other)`

`BusinessMonthEnd.copy`

`BusinessMonthEnd.isAnchored`

`BusinessMonthEnd.onOffset`

`BusinessMonthEnd.is_anchored`

`BusinessMonthEnd.is_on_offset`

`BusinessMonthEnd.__call__(*args, **kwargs)` Call self as a function.

`BusinessMonthEnd.is_month_start`

`BusinessMonthEnd.is_month_end`

`BusinessMonthEnd.is_quarter_start`

`BusinessMonthEnd.is_quarter_end`

`BusinessMonthEnd.is_year_start`

`BusinessMonthEnd.is_year_end`

`pandas.tseries.offsets.BusinessMonthEnd.apply`

`BusinessMonthEnd.apply()`

`pandas.tseries.offsets.BusinessMonthEnd.apply_index`

`BusinessMonthEnd.apply_index(other)`

`pandas.tseries.offsets.BusinessMonthEnd.copy`

`BusinessMonthEnd.copy()`
pandas.tseries.offsets.BusinessMonthEnd.isAnchored

BusinessMonthEnd.isAnchored()

pandas.tseries.offsets.BusinessMonthEnd.onOffset

BusinessMonthEnd.onOffset()

pandas.tseries.offsets.BusinessMonthEnd.is_anchored

BusinessMonthEnd.is_anchored()

pandas.tseries.offsets.BusinessMonthEnd.is_on_offset

BusinessMonthEnd.is_on_offset()

pandas.tseries.offsets.BusinessMonthEnd.is_month_start

BusinessMonthEnd.is_month_start()

pandas.tseries.offsets.BusinessMonthEnd.is_month_end

BusinessMonthEnd.is_month_end()

pandas.tseries.offsets.BusinessMonthEnd.is_quarter_start

BusinessMonthEnd.is_quarter_start()

pandas.tseries.offsets.BusinessMonthEnd.is_quarter_end

BusinessMonthEnd.is_quarter_end()

pandas.tseries.offsets.BusinessMonthEnd.is_year_start

BusinessMonthEnd.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessMonthEnd.is_year_end

BusinessMonthEnd.is_year_end()

3.7.9 BusinessMonthBegin

| BusinessMonthBegin | DateOffset of one month at the first business day. |

pandas.tseries.offsets.BusinessMonthBegin
class pandas.tseries.offsets.BusinessMonthBegin
    DateOffset of one month at the first business day.

Examples

```python
>>> from pandas.tseries.offsets import BMonthBegin
>>> ts=pd.Timestamp('2020-05-24 05:01:15')
>>> ts + BMonthBegin()
Timestamp('2020-06-01 05:01:15')
>>> ts + BMonthBegin(2)
Timestamp('2020-07-01 05:01:15')
>>> ts + BMonthBegin(-3)
Timestamp('2020-03-02 05:01:15')
```

Attributes

| base | Returns a copy of the calling offset object with n=1 and all other attributes equal. |

pandas.tseries.offsets.BusinessMonthBegin.base

BusinessMonthBegin.base
    Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>nanos</td>
<td></td>
</tr>
<tr>
<td>normalize</td>
<td></td>
</tr>
<tr>
<td>rule_code</td>
<td></td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>((*, **k)args)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.BusinessMonthBegin.__call__

BusinessMonthBegin.__call__\((*, **k\)args\)
Call self as a function.

pandas.tseries.offsets.BusinessMonthBegin.rollback

BusinessMonthBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.BusinessMonthBegin.rollforward

BusinessMonthBegin.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

Alias:

**BMonthBegin** alias of **pandas._libs.tslibs.offsets.BMonthBegin**

### 3.7. Date offsets

2339
**pandas.tseries.offsets.BMonthBegin**

alias of `pandas._libs.tslibs.offsets.BusinessMonthBegin`  

**Properties**

- `BusinessMonthBegin.freqstr`
- `BusinessMonthBegin.kwds`
- `BusinessMonthBegin.name`
- `BusinessMonthBegin.nanos`
- `BusinessMonthBegin.normalize`
- `BusinessMonthBegin.rule_code`
- `BusinessMonthBegin.n`

**pandas.tseries.offsets.BusinessMonthBegin.freqstr**

`BusinessMonthBegin.freqstr`

**pandas.tseries.offsets.BusinessMonthBegin.kwds**

`BusinessMonthBegin.kwds`

**pandas.tseries.offsets.BusinessMonthBegin.name**

`BusinessMonthBegin.name`

**pandas.tseries.offsets.BusinessMonthBegin.nanos**

`BusinessMonthBegin.nanos`
pandas.tseries.offsets.BusinessMonthBegin.normalize

BusinessMonthBegin.normalize

pandas.tseries.offsets.BusinessMonthBegin.rule_code

BusinessMonthBegin.rule_code

pandas.tseries.offsets.BusinessMonthBegin.n

BusinessMonthBegin.n

Methods

BusinessMonthBegin.apply

BusinessMonthBegin.apply_index(other)

BusinessMonthBegin.copy

BusinessMonthBegin.isAnchored

BusinessMonthBegin.onOffset

BusinessMonthBegin.is_anchored

BusinessMonthBegin.is_on_offset

BusinessMonthBegin.__call__(*args, **kwargs) Call self as a function.

BusinessMonthBegin.is_month_start

BusinessMonthBegin.is_month_end

BusinessMonthBegin.is_quarter_start

BusinessMonthBegin.is_quarter_end

BusinessMonthBegin.is_year_start

BusinessMonthBegin.is_year_end
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BusinessMonthBegin.apply
BusinessMonthBegin.apply()

pandas.tseries.offsets.BusinessMonthBegin.apply_index
BusinessMonthBegin.apply_index(other)

pandas.tseries.offsets.BusinessMonthBegin.copy
BusinessMonthBegin.copy()

pandas.tseries.offsets.BusinessMonthBegin.isAnchored
BusinessMonthBegin.isAnchored()

pandas.tseries.offsets.BusinessMonthBegin.onOffset
BusinessMonthBegin.onOffset()

pandas.tseries.offsets.BusinessMonthBegin.is_anchored
BusinessMonthBegin.is_anchored()

pandas.tseries.offsets.BusinessMonthBegin.is_on_offset
BusinessMonthBegin.is_on_offset()

pandas.tseries.offsets.BusinessMonthBegin.is_month_start
BusinessMonthBegin.is_month_start()

pandas.tseries.offsets.BusinessMonthBegin.is_month_end
BusinessMonthBegin.is_month_end()
pandas.tseries.offsets.BusinessMonthBegin.is_quarter_start

BusinessMonthBegin.is_quarter_start()

pandas.tseries.offsets.BusinessMonthBegin.is_quarter_end

BusinessMonthBegin.is_quarter_end()

pandas.tseries.offsets.BusinessMonthBegin.is_year_start

BusinessMonthBegin.is_year_start()

pandas.tseries.offsets.BusinessMonthBegin.is_year_end

BusinessMonthBegin.is_year_end()

3.7.10 CustomBusinessMonthEnd

CustomBusinessMonthEnd

Attributes

pandas.tseries.offsets.CustomBusinessMonthEnd

class pandas.tseries.offsets.CustomBusinessMonthEnd

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Returns a copy of the calling offset object with n=1 and all other attributes equal.</td>
</tr>
<tr>
<td>cbday_roll</td>
<td>Define default roll function to be called in apply method.</td>
</tr>
<tr>
<td>month_roll</td>
<td>Define default roll function to be called in apply method.</td>
</tr>
<tr>
<td>offset</td>
<td>Alias for self._offset.</td>
</tr>
</tbody>
</table>
**pandas.tseries.offsets.CustomBusinessMonthEnd.base**

CustomBusinessMonthEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

**pandas.tseries.offsets.CustomBusinessMonthEnd.cbdy_roll**

CustomBusinessMonthEnd.cbdy_roll

Define default roll function to be called in apply method.

**pandas.tseries.offsets.CustomBusinessMonthEnd.month_roll**

CustomBusinessMonthEnd.month_roll

Define default roll function to be called in apply method.

**pandas.tseries.offsets.CustomBusinessMonthEnd.offset**

CustomBusinessMonthEnd.offset

Alias for self._offset.

<table>
<thead>
<tr>
<th>calendar</th>
<th>freqstr</th>
<th>holidays</th>
<th>kwds</th>
<th>m_offset</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
<th>weekmask</th>
</tr>
</thead>
</table>

**Methods**

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, **kwargs)</th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>
pandas.tseries.offsets.CustomBusinessMonthEnd.__call__

CustomBusinessMonthEnd.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.CustomBusinessMonthEnd.rollback

CustomBusinessMonthEnd.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.CustomBusinessMonthEnd.rollforward

CustomBusinessMonthEnd.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>

Alias:

| CBMonthEnd | alias of pandas._libs.tslibs.offsets.CustomBusinessMonthEnd |
pandas.tseries.offsets.CBMonthEnd

alias of pandas._libs.tslibs.offsets.CustomBusinessMonthEnd

Properties

- CustomBusinessMonthEnd.freqstr
- CustomBusinessMonthEnd.kwds
- CustomBusinessMonthEnd.m_offset
- CustomBusinessMonthEnd.name
- CustomBusinessMonthEnd.nanos
- CustomBusinessMonthEnd.normalize
- CustomBusinessMonthEnd.rule_code
- CustomBusinessMonthEnd.n
- CustomBusinessMonthEnd.weekmask
- CustomBusinessMonthEnd.calendar
- CustomBusinessMonthEnd.holidays

pandas.tseries.offsets.CustomBusinessMonthEnd.freqstr

CustomBusinessMonthEnd.freqstr

pandas.tseries.offsets.CustomBusinessMonthEnd.kwds

CustomBusinessMonthEnd.kwds

pandas.tseries.offsets.CustomBusinessMonthEnd.m_offset

CustomBusinessMonthEnd.m_offset
pandas.tseries.offsets.CustomBusinessMonthEnd.name

CustomBusinessMonthEnd.name

pandas.tseries.offsets.CustomBusinessMonthEnd.nanos

CustomBusinessMonthEnd.nanos

pandas.tseries.offsets.CustomBusinessMonthEnd.normalize

CustomBusinessMonthEnd.normalize

pandas.tseries.offsets.CustomBusinessMonthEnd.rule_code

CustomBusinessMonthEnd.rule_code

pandas.tseries.offsets.CustomBusinessMonthEnd.n

CustomBusinessMonthEnd.n

pandas.tseries.offsets.CustomBusinessMonthEnd.weekmask

CustomBusinessMonthEnd.weekmask

pandas.tseries.offsets.CustomBusinessMonthEnd.calendar

CustomBusinessMonthEnd.calendar

pandas.tseries.offsets.CustomBusinessMonthEnd.holidays

CustomBusinessMonthEnd.holidays

3.7. Date offsets
Methods

CustomBusinessMonthEnd.apply

CustomBusinessMonthEnd.apply_index(\texttt{other})

CustomBusinessMonthEnd.copy

CustomBusinessMonthEnd.isAnchored

CustomBusinessMonthEnd.onOffset

CustomBusinessMonthEnd.is_anchored

CustomBusinessMonthEnd.is_on_offset

CustomBusinessMonthEnd.__call__(*\texttt{args}, **\texttt{kwargs})  \textit{Call self as a function.}

CustomBusinessMonthEnd.is_month_start

CustomBusinessMonthEnd.is_month_end

CustomBusinessMonthEnd.is_quarter_start

CustomBusinessMonthEnd.is_quarter_end

CustomBusinessMonthEnd.is_year_start

CustomBusinessMonthEnd.is_year_end

\texttt{pandas.tseries.offsets.CustomBusinessMonthEnd.apply}

\texttt{CustomBusinessMonthEnd.apply()}

\texttt{pandas.tseries.offsets.CustomBusinessMonthEnd.apply_index}

\texttt{CustomBusinessMonthEnd.apply_index(\texttt{other})}

\texttt{pandas.tseries.offsets.CustomBusinessMonthEnd.copy}

\texttt{CustomBusinessMonthEnd.copy()}
pandas.tseries.offsets.CustomBusinessMonthEnd.isAnchored

CustomBusinessMonthEnd.isAnchored()

pandas.tseries.offsets.CustomBusinessMonthEnd.onOffset

CustomBusinessMonthEnd.onOffset()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_anchored

CustomBusinessMonthEnd.is_anchored()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_on_offset

CustomBusinessMonthEnd.is_on_offset()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_month_start

CustomBusinessMonthEnd.is_month_start()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_month_end

CustomBusinessMonthEnd.is_month_end()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_quarter_start

CustomBusinessMonthEnd.is_quarter_start()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_quarter_end

CustomBusinessMonthEnd.is_quarter_end()

pandas.tseries.offsets.CustomBusinessMonthEnd.is_year_start

CustomBusinessMonthEnd.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

CustomBusinessMonthEnd.is_year_end

3.7.11 CustomBusinessMonthBegin

CustomBusinessMonthBegin

Attributes

pandas.tseries.offsets.CustomBusinessMonthBegin
class pandas.tseries.offsets.CustomBusinessMonthBegin

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbday_roll</td>
<td>Define default roll function to be called in apply method.</td>
</tr>
<tr>
<td>month_roll</td>
<td>Define default roll function to be called in apply method.</td>
</tr>
<tr>
<td>offset</td>
<td>Alias for self._offset.</td>
</tr>
</tbody>
</table>

CustomBusinessMonthBegin.base

CustomBusinessMonthBegin.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

CustomBusinessMonthBegin.cbd_day_roll

CustomBusinessMonthBegin.cbd_day_roll

Define default roll function to be called in apply method.

CustomBusinessMonthBegin.month_roll

CustomBusinessMonthBegin.month_roll

Define default roll function to be called in apply method.
pandas.tseries.offsets.CustomBusinessMonthBegin.offset

CustomBusinessMonthBegin.offset
Alias for self._offset.

<table>
<thead>
<tr>
<th>calendar</th>
<th>freqstr</th>
<th>holidays</th>
<th>kwds</th>
<th>m_offset</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
<th>weekmask</th>
</tr>
</thead>
</table>

Methods

__call__(*args, **kwargs) Call self as a function.

rollback
Roll provided date backward to next offset only if not on offset.

rollforward
Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessMonthBegin.__call__

CustomBusinessMonthBegin.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.CustomBusinessMonthBegin.rollback

CustomBusinessMonthBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.
pandas.tseries.offsets.CustomBusinessMonthBegin.rollforward

CustomBusinessMonthBegin.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

**TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
</tr>
</tbody>
</table>

Alias:

| CBMonthBegin | alias of pandas._libs.tslibs.offsets.CustomBusinessMonthBegin |

pandas.tseries.offsets.CBMonthBegin

pandas.tseries.offsets.CBMonthBegin

alias of pandas._libs.tslibs.offsets.CustomBusinessMonthBegin
Properties

- `CustomBusinessMonthBegin.freqstr`
- `CustomBusinessMonthBegin.kwds`
- `CustomBusinessMonthBegin.m_offset`
- `CustomBusinessMonthBegin.name`
- `CustomBusinessMonthBegin.nanos`
- `CustomBusinessMonthBegin.normalize`
- `CustomBusinessMonthBegin.rule_code`
- `CustomBusinessMonthBegin.n`
- `CustomBusinessMonthBegin.weekmask`
- `CustomBusinessMonthBegin.calendar`
- `CustomBusinessMonthBegin.holidays`

```python
pandas.tseries.offsets.CustomBusinessMonthBegin.freqstr
```

```python
CustomBusinessMonthBegin.freqstr
```

```python
pandas.tseries.offsets.CustomBusinessMonthBegin.kwds
```

```python
CustomBusinessMonthBegin.kwds
```

```python
pandas.tseries.offsets.CustomBusinessMonthBegin.m_offset
```

```python
CustomBusinessMonthBegin.m_offset
```

```python
pandas.tseries.offsets.CustomBusinessMonthBegin.name
```

```python
CustomBusinessMonthBegin.name
```
pandas.tseries.offsets.CustomBusinessMonthBegin.nanos

CustomBusinessMonthBegin.nanos

pandas.tseries.offsets.CustomBusinessMonthBegin.normalize

CustomBusinessMonthBegin.normalize

pandas.tseries.offsets.CustomBusinessMonthBegin.rule_code

CustomBusinessMonthBegin.rule_code

pandas.tseries.offsets.CustomBusinessMonthBegin.n

CustomBusinessMonthBegin.n

pandas.tseries.offsets.CustomBusinessMonthBegin.weekmask

CustomBusinessMonthBegin.weekmask

pandas.tseries.offsets.CustomBusinessMonthBegin.calendar

CustomBusinessMonthBegin.calendar

pandas.tseries.offsets.CustomBusinessMonthBegin.holidays

CustomBusinessMonthBegin.holidays
Methods

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomBusinessMonthBegin.apply</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.apply_index(other)</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.copy</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.isAnchored</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.onOffset</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_anchored</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_on_offset</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.<strong>call</strong>(*args, ...)</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_month_start</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_month_end</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_quarter_start</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_quarter_end</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_year_start</td>
</tr>
<tr>
<td>CustomBusinessMonthBegin.is_year_end</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.CustomBusinessMonthBegin.apply

CustomBusinessMonthBegin.apply()

pandas.tseries.offsets.CustomBusinessMonthBegin.apply_index

CustomBusinessMonthBegin.apply_index(other)

pandas.tseries.offsets.CustomBusinessMonthBegin.copy

CustomBusinessMonthBegin.copy()
pandas.tseries.offsets.CustomBusinessMonthBegin.isAnchored

CustomBusinessMonthBegin.isAnchored()

pandas.tseries.offsets.CustomBusinessMonthBegin.onOffset

CustomBusinessMonthBegin.onOffset()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_anchored

CustomBusinessMonthBegin.is_anchored()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_on_offset

CustomBusinessMonthBegin.is_on_offset()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_month_start

CustomBusinessMonthBegin.is_month_start()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_month_end

CustomBusinessMonthBegin.is_month_end()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_quarter_start

CustomBusinessMonthBegin.is_quarter_start()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_quarter_end

CustomBusinessMonthBegin.is_quarter_end()

pandas.tseries.offsets.CustomBusinessMonthBegin.is_year_start

CustomBusinessMonthBegin.is_year_start()
**pandas.tseries.offsets.CustomBusinessMonthBegin.is_year_end**

CustomBusinessMonthBegin.is_year_end()

## 3.7.12 SemiMonthEnd

<table>
<thead>
<tr>
<th><strong>SemiMonthEnd</strong></th>
<th>Two DateOffset's per month repeating on the last day of the month and day_of_month.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.SemiMonthEnd**

**class** pandas.tseries.offsets.SemiMonthEnd

Two DateOffset’s per month repeating on the last day of the month and day_of_month.

**Parameters**

- **n** [int]
- **normalize** [bool, default False]
- **day_of_month** [int, {1, 3,…,27}, default 15]

**Attributes**

<table>
<thead>
<tr>
<th><strong>base</strong></th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.SemiMonthEnd.base**

SemiMonthEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>day_of_month</th>
</tr>
</thead>
<tbody>
<tr>
<td>freqstr</td>
</tr>
<tr>
<td>kwds</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>nanos</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>rule_code</td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.SemiMonthEnd.__call__

SemiMonthEnd.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.SemiMonthEnd.rollback

SemiMonthEnd.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.SemiMonthEnd.rollforward

SemiMonthEnd.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
</tr>
<tr>
<td>apply_index</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>isAnchored</td>
</tr>
<tr>
<td>is_anchored</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
</tr>
<tr>
<td>is_on_offset</td>
</tr>
<tr>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
</tr>
<tr>
<td>is_year_end</td>
</tr>
<tr>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
</tr>
</tbody>
</table>
## Properties

- `SemiMonthEnd.freqstr`
- `SemiMonthEnd.kwds`
- `SemiMonthEnd.name`
- `SemiMonthEnd.nanos`
- `SemiMonthEnd.normalize`
- `SemiMonthEnd.rule_code`
- `SemiMonthEnd.n`
- `SemiMonthEnd.day_of_month`

### pandas.tseries.offsets.SemiMonthEnd.freqstr

`SemiMonthEnd.freqstr`

### pandas.tseries.offsets.SemiMonthEnd.kwds

`SemiMonthEnd.kwds`

### pandas.tseries.offsets.SemiMonthEnd.name

`SemiMonthEnd.name`

### pandas.tseries.offsets.SemiMonthEnd.nanos

`SemiMonthEnd.nanos`

### pandas.tseries.offsets.SemiMonthEnd.normalize

`SemiMonthEnd.normalize`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.SemiMonthEnd.rule_code

SemiMonthEnd.rule_code

pandas.tseries.offsets.SemiMonthEnd.n

SemiMonthEnd.n

pandas.tseries.offsets.SemiMonthEnd.day_of_month

SemiMonthEnd.day_of_month

Methods

SemiMonthEnd.apply

SemiMonthEnd.apply_index(other)

SemiMonthEnd.copy

SemiMonthEnd.isAnchored

SemiMonthEnd.onOffset

SemiMonthEnd.is_anchored

SemiMonthEnd.is_on_offset

SemiMonthEnd.__call__(*args, **kwargs) Call self as a function.

SemiMonthEnd.is_month_start

SemiMonthEnd.is_month_end

SemiMonthEnd.is_quarter_start

SemiMonthEnd.is_quarter_end

SemiMonthEnd.is_year_start

SemiMonthEnd.is_year_end
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.SemiMonthEnd.apply

SemiMonthEnd.apply()

pandas.tseries.offsets.SemiMonthEnd.apply_index

SemiMonthEnd.apply_index(other)

pandas.tseries.offsets.SemiMonthEnd.copy

SemiMonthEnd.copy()

pandas.tseries.offsets.SemiMonthEnd.isAnchored

SemiMonthEnd.isAnchored()

pandas.tseries.offsets.SemiMonthEnd.onOffset

SemiMonthEnd.onOffset()

pandas.tseries.offsets.SemiMonthEnd.is_anchored

SemiMonthEnd.is_anchored()

pandas.tseries.offsets.SemiMonthEnd.is_on_offset

SemiMonthEnd.is_on_offset()

pandas.tseries.offsets.SemiMonthEnd.is_month_start

SemiMonthEnd.is_month_start()

pandas.tseries.offsets.SemiMonthEnd.is_month_end

SemiMonthEnd.is_month_end()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.SemiMonthEnd.is_quarter_start
SemiMonthEnd.is_quarter_start()

pandas.tseries.offsets.SemiMonthEnd.is_quarter_end
SemiMonthEnd.is_quarter_end()

pandas.tseries.offsets.SemiMonthEnd.is_year_start
SemiMonthEnd.is_year_start()

pandas.tseries.offsets.SemiMonthEnd.is_year_end
SemiMonthEnd.is_year_end()

3.7.13 SemiMonthBegin

<table>
<thead>
<tr>
<th>SemiMonthBegin</th>
<th>Two DateOffset’s per month repeating on the first day of the month and day_of_month.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.SemiMonthBegin
class pandas.tseries.offsets.SemiMonthBegin
Two DateOffset’s per month repeating on the first day of the month and day_of_month.

Parameters

- n [int]
- normalize [bool, default False]
- day_of_month [int, {2, 3,...,27}, default 15]

Attributes

- base
  Returns a copy of the calling offset object with n=1 and all other attributes equal.
pandas.tseries.offsets.SemiMonthBegin.base

SemiMonthBegin.base
Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>day_of_month</th>
<th>freqstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
<td>n</td>
</tr>
<tr>
<td>name</td>
<td>nanos</td>
</tr>
<tr>
<td>normalize</td>
<td>rule_code</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, **kwargs)</th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.SemiMonthBegin.__call__

SemiMonthBegin.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.SemiMonthBegin.rollback

SemiMonthBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.SemiMonthBegin.rollforward

SemiMonthBegin.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.
### Properties

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SemiMonthBegin.freqstr</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.kwds</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.name</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.nanos</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.normalize</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.rule_code</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.n</code></td>
</tr>
<tr>
<td><code>SemiMonthBegin.day_of_month</code></td>
</tr>
</tbody>
</table>

### pandas.tseries.offsets.SemiMonthBegin.freqstr

SemiMonthBegin.freqstr

### pandas.tseries.offsets.SemiMonthBegin.kwds

SemiMonthBegin.kwds
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.SemiMonthBegin.name
SemiMonthBegin.name

pandas.tseries.offsets.SemiMonthBegin.nanos
SemiMonthBegin.nanos

pandas.tseries.offsets.SemiMonthBegin.normalize
SemiMonthBegin.normalize

pandas.tseries.offsets.SemiMonthBegin.rule_code
SemiMonthBegin.rule_code

pandas.tseries.offsets.SemiMonthBegin.n
SemiMonthBegin.n

pandas.tseries.offsets.SemiMonthBegin.day_of_month
SemiMonthBegin.day_of_month
### Methods

- `SemiMonthBegin.apply`
- `SemiMonthBegin.apply_index(other)`
- `SemiMonthBegin.copy`
- `SemiMonthBegin.isAnchored`
- `SemiMonthBegin.onOffset`
- `SemiMonthBegin.is_anchored`
- `SemiMonthBegin.is_on_offset`
- `SemiMonthBegin.__call__(args, **kwargs)`  
  Call self as a function.
- `SemiMonthBegin.is_month_start`
- `SemiMonthBegin.is_month_end`
- `SemiMonthBegin.is_quarter_start`
- `SemiMonthBegin.is_quarter_end`
- `SemiMonthBegin.is_year_start`
- `SemiMonthBegin.is_year_end`

---

**pandas.tseries.offsets.SemiMonthBegin.apply**

SemiMonthBegin.apply()

**pandas.tseries.offsets.SemiMonthBegin.apply_index**

SemiMonthBegin.apply_index(other)

**pandas.tseries.offsets.SemiMonthBegin.copy**

SemiMonthBegin.copy()
pandas.tseries.offsets.SemiMonthBegin.isAnchored

SemiMonthBegin.isAnchored()

pandas.tseries.offsets.SemiMonthBegin.onOffset

SemiMonthBegin.onOffset()

pandas.tseries.offsets.SemiMonthBegin.is_anchored

SemiMonthBegin.is_anchored()

pandas.tseries.offsets.SemiMonthBegin.is_on_offset

SemiMonthBegin.is_on_offset()

pandas.tseries.offsets.SemiMonthBegin.is_month_start

SemiMonthBegin.is_month_start()

pandas.tseries.offsets.SemiMonthBegin.is_month_end

SemiMonthBegin.is_month_end()

pandas.tseries.offsets.SemiMonthBegin.is_quarter_start

SemiMonthBegin.is_quarter_start()

pandas.tseries.offsets.SemiMonthBegin.is_quarter_end

SemiMonthBegin.is_quarter_end()

pandas.tseries.offsets.SemiMonthBegin.is_year_start

SemiMonthBegin.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.SemiMonthBegin.is_year_end

SemiMonthBegin.is_year_end()

3.7.14 Week

---

**Week**

Weekly offset.

---

**pandas.tseries.offsets.Week**

*class* pandas.tseries.offsets.Week

Weekly offset.

**Parameters**

- **weekday** [int or None, default None] Always generate specific day of week. 0 for Monday.

**Attributes**

- **base** Returns a copy of the calling offset object with n=1 and all other attributes equal.

---

**pandas.tseries.offsets.Week.base**

Week.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

---

<table>
<thead>
<tr>
<th>freqstr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>nanos</td>
<td></td>
</tr>
<tr>
<td>normalize</td>
<td></td>
</tr>
<tr>
<td>rule_code</td>
<td></td>
</tr>
<tr>
<td>weekday</td>
<td></td>
</tr>
</tbody>
</table>

**Methods**

- **__call__**(args, **kwargs) Call self as a function.

- **rollback** Roll provided date backward to next offset only if not on offset.

- **rollforward** Roll provided date forward to next offset only if not on offset.
pandas.tseries.offsets.Week.__call__

Week.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.Week.rollback

Week.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Week.rollforward

Week.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>isAnchored</td>
<td>is_anchored</td>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Properties

- Week.freqstr
- Week.kwds
- Week.name
- Week.nanos
- Week.normalize
- Week.rule_code
- Week.n
- Week.weekday

pandas.tseries.offsets.Week.freqstr

Week.freqstr

pandas.tseries.offsets.Week.kwds

Week.kwds

pandas.tseries.offsets.Week.name

Week.name

pandas.tseries.offsets.Week.nanos

Week.nanos

pandas.tseries.offsets.Week.normalize

Week.normalize
pandas.tseries.offsets.Week.rule_code

Week.rule_code

pandas.tseries.offsets.Week.n

Week.n

pandas.tseries.offsets.Week.weekday

Week.weekday

Methods

Week.apply

Week.apply_index(other)

Week.copy

Week.isAnchored

Week.onOffset

Week.is_anchored

Week.is_on_offset

Week.__call__(*args, **kwargs)  Call self as a function.

Week.is_month_start

Week.is_month_end

Week.is_quarter_start

Week.is_quarter_end

Week.is_year_start

Week.is_year_end
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Week.apply

Week.apply()

pandas.tseries.offsets.Week.apply_index

Week.apply_index(other)

pandas.tseries.offsets.Week.copy

Week.copy()

pandas.tseries.offsets.Week.isAnchored

Week.isAnchored()

pandas.tseries.offsets.Week.onOffset

Week.onOffset()

pandas.tseries.offsets.Week.is_anchored

Week.is_anchored()

pandas.tseries.offsets.Week.is_on_offset

Week.is_on_offset()

pandas.tseries.offsets.Week.is_month_start

Week.is_month_start()

pandas.tseries.offsets.Week.is_month_end

Week.is_month_end()
pandas.tseries.offsets.Week.is_quarter_start

Week.is_quarter_start()

pandas.tseries.offsets.Week.is_quarter_end

Week.is_quarter_end()

pandas.tseries.offsets.Week.is_year_start

Week.is_year_start()

pandas.tseries.offsets.Week.is_year_end

Week.is_year_end()

3.7.15 WeekOfMonth

| WeekOfMonth | Describes monthly dates like “the Tuesday of the 2nd week of each month”.
|-------------|---------------------------------------------------|

class pandas.tseries.offsets.WeekOfMonth

Describes monthly dates like “the Tuesday of the 2nd week of each month”.

Parameters

- n [int]
- week [int {0, 1, 2, 3, ...}, default 0] A specific integer for the week of the month. e.g. 0 is 1st week of month, 1 is the 2nd week, etc.
- weekday [int {0, 1, ..., 6}, default 0] A specific integer for the day of the week.
  - 0 is Monday
  - 1 is Tuesday
  - 2 is Wednesday
  - 3 is Thursday
  - 4 is Friday
  - 5 is Saturday
  - 6 is Sunday.
Attributes

### pandas.tseries.offsets.WeekOfMonth.base

`WeekOfMonth.base`

Returns a copy of the calling offset object with `n=1` and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th>kwds</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
<th>week</th>
<th>weekday</th>
</tr>
</thead>
</table>

Methods

### pandas.tseries.offsets.WeekOfMonth.__call__

`WeekOfMonth.__call__(*args, **kwargs)`

Call self as a function.

### pandas.tseries.offsets.WeekOfMonth.rollback

`WeekOfMonth.rollback()`

Roll provided date backward to next offset only if not on offset.

Returns

`TimeStamp` Rolled timestamp if not on offset, otherwise unchanged timestamp.

### pandas.tseries.offsets.WeekOfMonth.rollforward

`WeekOfMonth.rollforward()`

Roll provided date forward to next offset only if not on offset.
pandas.tseries.offsets.WeekOfMonth.rollforward

WeekOfMonth.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
</tr>
</tbody>
</table>

Properties

WeekOfMonth.freqstr

WeekOfMonth.kwds

WeekOfMonth.name

WeekOfMonth.nanos

WeekOfMonth.normalize

WeekOfMonth.rule_code

WeekOfMonth.n

WeekOfMonth.week
pandas.tseries.offsets.WeekOfMonth.freqstr

WeekOfMonth.freqstr

pandas.tseries.offsets.WeekOfMonth.kwds

WeekOfMonth.kwds

pandas.tseries.offsets.WeekOfMonth.name

WeekOfMonth.name

pandas.tseries.offsets.WeekOfMonth.nanos

WeekOfMonth.nanos

pandas.tseries.offsets.WeekOfMonth.normalize

WeekOfMonth.normalize

pandas.tseries.offsets.WeekOfMonth.rule_code

WeekOfMonth.rule_code

pandas.tseries.offsets.WeekOfMonth.n

WeekOfMonth.n

pandas.tseries.offsets.WeekOfMonth.week

WeekOfMonth.week
Methods

```python
WeekOfMonth.apply

WeekOfMonth.apply_index(other)

WeekOfMonth.copy

WeekOfMonth.isAnchored

WeekOfMonth.onOffset

WeekOfMonth.is_anchored

WeekOfMonth.is_on_offset

WeekOfMonth.__call__(*args, **kwargs) Call self as a function.

WeekOfMonth.weekday

WeekOfMonth.is_month_start

WeekOfMonth.is_month_end

WeekOfMonth.is_quarter_start

WeekOfMonth.is_quarter_end

WeekOfMonth.is_year_start

WeekOfMonth.is_year_end
```

```python
pandas.tseries.offsets.WeekOfMonth.apply

WeekOfMonth.apply()
```
pandas.tseries.offsets.WeekOfMonth.apply_index

WeekOfMonth.apply_index(other)

pandas.tseries.offsets.WeekOfMonth.copy

WeekOfMonth.copy()

pandas.tseries.offsets.WeekOfMonth.isAnchored

WeekOfMonth.isAnchored()

pandas.tseries.offsets.WeekOfMonth.onOffset

WeekOfMonth.onOffset()

pandas.tseries.offsets.WeekOfMonth.is_anchored

WeekOfMonth.is_anchored()

pandas.tseries.offsets.WeekOfMonth.is_on_offset

WeekOfMonth.is_on_offset()

pandas.tseries.offsets.WeekOfMonth.weekday

WeekOfMonth.weekday

pandas.tseries.offsets.WeekOfMonth.is_month_start

WeekOfMonth.is_month_start()

pandas.tseries.offsets.WeekOfMonth.is_month_end

WeekOfMonth.is_month_end()
3.7. Date offsets

pandas.tseries.offsets.WeekOfMonth.is_quarter_start

WeekOfMonth.is_quarter_start()

pandas.tseries.offsets.WeekOfMonth.is_quarter_end

WeekOfMonth.is_quarter_end()

pandas.tseries.offsets.WeekOfMonth.is_year_start

WeekOfMonth.is_year_start()

pandas.tseries.offsets.WeekOfMonth.is_year_end

WeekOfMonth.is_year_end()

3.7.16 LastWeekOfMonth

<table>
<thead>
<tr>
<th>LastWeekOfMonth</th>
<th>Describes monthly dates in last week of month like &quot;the last Tuesday of each month&quot;.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.LastWeekOfMonth

class pandas.tseries.offsets.LastWeekOfMonth

    Describes monthly dates in last week of month like “the last Tuesday of each month”.

    Parameters

    n [int, default 1]

    weekday [int {0, 1, …, 6}, default 0] A specific integer for the day of the week.

    • 0 is Monday
    • 1 is Tuesday
    • 2 is Wednesday
    • 3 is Thursday
    • 4 is Friday
    • 5 is Saturday
    • 6 is Sunday.
**Attributes**

<table>
<thead>
<tr>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a copy of the calling offset object with n=1 and all other attributes equal.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.LastWeekOfMonth.base**

LastWeekOfMonth.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>nanos</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>rule_code</td>
</tr>
<tr>
<td>week</td>
</tr>
<tr>
<td>weekday</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, **kwargs)</th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.LastWeekOfMonth.__call__**

LastWeekOfMonth.__call__(*args, **kwargs)

Call self as a function.

**pandas.tseries.offsets.LastWeekOfMonth.rollback**

LastWeekOfMonth.rollback()

Roll provided date backward to next offset only if not on offset.

**Returns**

**Timestamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.
pandas.tseries.offsets.LastWeekOfMonth.rollforward

LastWeekOfMonth.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

Properties

LastWeekOfMonth.freqstr
LastWeekOfMonth.kwds
LastWeekOfMonth.name
LastWeekOfMonth.nanos
LastWeekOfMonth.normalize
LastWeekOfMonth.rule_code
LastWeekOfMonth.n
LastWeekOfMonth.weekday
LastWeekOfMonth.week
pandas.tseries.offsets.LastWeekOfMonth.freqstr

LastWeekOfMonth.freqstr

pandas.tseries.offsets.LastWeekOfMonth.kwds

LastWeekOfMonth.kwds

pandas.tseries.offsets.LastWeekOfMonth.name

LastWeekOfMonth.name

pandas.tseries.offsets.LastWeekOfMonth.nanos

LastWeekOfMonth.nanos

pandas.tseries.offsets.LastWeekOfMonth.normalize

LastWeekOfMonth.normalize

pandas.tseries.offsets.LastWeekOfMonth.rule_code

LastWeekOfMonth.rule_code

pandas.tseries.offsets.LastWeekOfMonth.n

LastWeekOfMonth.n

pandas.tseries.offsets.LastWeekOfMonth.weekday

LastWeekOfMonth.weekday

pandas.tseries.offsets.LastWeekOfMonth.week

LastWeekOfMonth.week
Methods

LastWeekOfMonth.apply

LastWeekOfMonth.apply_index(other)

LastWeekOfMonth.copy

LastWeekOfMonth.isAnchored

LastWeekOfMonth.onOffset

LastWeekOfMonth.is_anchored

LastWeekOfMonth.is_on_offset

LastWeekOfMonth.__call__(*args, **kwargs) Call self as a function.

LastWeekOfMonth.is_month_start

LastWeekOfMonth.is_month_end

LastWeekOfMonth.is_quarter_start

LastWeekOfMonth.is_quarter_end

LastWeekOfMonth.is_year_start

LastWeekOfMonth.is_year_end

pandas.tseries.offsets.LastWeekOfMonth.apply

LastWeekOfMonth.apply()

pandas.tseries.offsets.LastWeekOfMonth.apply_index

LastWeekOfMonth.apply_index(other)

pandas.tseries.offsets.LastWeekOfMonth.copy

LastWeekOfMonth.copy()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.LastWeekOfMonth.isAnchored

LastWeekOfMonth.isAnchored()

pandas.tseries.offsets.LastWeekOfMonth.onOffset

LastWeekOfMonth.onOffset()

pandas.tseries.offsets.LastWeekOfMonth.is_anchored

LastWeekOfMonth.is_anchored()

pandas.tseries.offsets.LastWeekOfMonth.is_on_offset

LastWeekOfMonth.is_on_offset()

pandas.tseries.offsets.LastWeekOfMonth.is_month_start

LastWeekOfMonth.is_month_start()

pandas.tseries.offsets.LastWeekOfMonth.is_month_end

LastWeekOfMonth.is_month_end()

pandas.tseries.offsets.LastWeekOfMonth.is_quarter_start

LastWeekOfMonth.is_quarter_start()

pandas.tseries.offsets.LastWeekOfMonth.is_quarter_end

LastWeekOfMonth.is_quarter_end()

pandas.tseries.offsets.LastWeekOfMonth.is_year_start

LastWeekOfMonth.is_year_start()
pandas.tseries.offsets.LastWeekOfMonth.is_year_end

LastWeekOfMonth.is_year_end()

3.7.17 BQuarterEnd

<table>
<thead>
<tr>
<th>BQuarterEnd</th>
<th>DateOffset increments between the last business day of each Quarter.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BQuarterEnd

class pandas.tseries.offsets.BQuarterEnd

DateOffset increments between the last business day of each Quarter.

startingMonth = 1 corresponds to dates like 1/31/2007, 4/30/2007, ... startingMonth = 2 corresponds to dates like 2/28/2007, 5/31/2007, ... startingMonth = 3 corresponds to dates like 3/30/2007, 6/29/2007, ...

Examples

```python
>>> from pandas.tseries.offsets import BQuarterEnd
>>> ts = pd.Timestamp('2020-05-24 05:01:15')
>>> ts + BQuarterEnd()
Timestamp('2020-06-30 05:01:15')
>>> ts + BQuarterEnd(2)
Timestamp('2020-09-30 05:01:15')
>>> ts + BQuarterEnd(1, startingMonth=2)
Timestamp('2020-05-29 05:01:15')
>>> ts + BQuarterEnd(startingMonth=2)
Timestamp('2020-05-29 05:01:15')
```

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BQuarterEnd.base

BQuarterEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.BQuarterEnd.__call__**

BQuarterEnd.__call__(*args, **kwargs)

Call self as a function.

**pandas.tseries.offsets.BQuarterEnd.rollback**

BQuarterEnd.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

**TimeStamp**  Rolled timestamp if not on offset, otherwise unchanged timestamp.

**pandas.tseries.offsets.BQuarterEnd.rollforward**

BQuarterEnd.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

**TimeStamp**  Rolled timestamp if not on offset, otherwise unchanged timestamp.
Properties

- `BQuarterEnd.freqstr`
- `BQuarterEnd.kwds`
- `BQuarterEnd.name`
- `BQuarterEnd.nanos`
- `BQuarterEnd.normalize`
- `BQuarterEnd.rule_code`
- `BQuarterEnd.n`
- `BQuarterEnd.startingMonth`

`pandas.tseries.offsets.BQuarterEnd.freqstr`

`BQuarterEnd.freqstr`

`pandas.tseries.offsets.BQuarterEnd.kwds`

`BQuarterEnd.kwds`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BQuarterEnd.name

BQuarterEnd.name

pandas.tseries.offsets.BQuarterEnd.nanos

BQuarterEnd.nanos

pandas.tseries.offsets.BQuarterEnd.normalize

BQuarterEnd.normalize

pandas.tseries.offsets.BQuarterEnd.rule_code

BQuarterEnd.rule_code

pandas.tseries.offsets.BQuarterEnd.n

BQuarterEnd.n

pandas.tseries.offsets.BQuarterEnd.startingMonth

BQuarterEnd.startingMonth
Methods

BQuarterEnd.apply
BQuarterEnd.apply_index(other)
BQuarterEnd.copy
BQuarterEnd.isAnchored
BQuarterEnd.onOffset
BQuarterEnd.is_anchored
BQuarterEnd.is_on_offset
BQuarterEnd.__call__(*args, **kwargs) Call self as a function.
BQuarterEnd.is_month_start
BQuarterEnd.is_month_end
BQuarterEnd.is_quarter_start
BQuarterEnd.is_quarter_end
BQuarterEnd.is_year_start
BQuarterEnd.is_year_end

pandas.tseries_offsets.BQuarterEnd.apply
BQuarterEnd.apply()

pandas.tseries_offsets.BQuarterEnd.apply_index
BQuarterEnd.apply_index(other)

pandas.tseries_offsets.BQuarterEnd.copy
BQuarterEnd.copy()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BQuarterEnd.isAnchored
BQuarterEnd.isAnchored()

pandas.tseries.offsets.BQuarterEnd.onOffset
BQuarterEnd.onOffset()

pandas.tseries.offsets.BQuarterEnd.is_anchored
BQuarterEnd.is_anchored()

pandas.tseries.offsets.BQuarterEnd.is_on_offset
BQuarterEnd.is_on_offset()

pandas.tseries.offsets.BQuarterEnd.is_month_start
BQuarterEnd.is_month_start()

pandas.tseries.offsets.BQuarterEnd.is_month_end
BQuarterEnd.is_month_end()

pandas.tseries.offsets.BQuarterEnd.is_quarter_start
BQuarterEnd.is_quarter_start()

pandas.tseries.offsets.BQuarterEnd.is_quarter_end
BQuarterEnd.is_quarter_end()

pandas.tseries.offsets.BQuarterEnd.is_year_start
BQuarterEnd.is_year_start()
pandas.tseries.offsets.BQuarterEnd.is_year_end

BQuarterEnd.is_year_end()

3.7.18 BQuarterBegin

<table>
<thead>
<tr>
<th>BQuarterBegin</th>
<th>DateOffset increments between the first business day of each Quarter.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BQuarterBegin

```
class pandas.tseries.offsets.BQuarterBegin
    DateOffset increments between the first business day of each Quarter.
    startingMonth = 1 corresponds to dates like 1/01/2007, 4/01/2007, ... startingMonth = 2 corresponds to dates like 2/01/2007, 5/01/2007, ... startingMonth = 3 corresponds to dates like 3/01/2007, 6/01/2007, ...

Examples
```
```python
>>> from pandas.tseries.offsets import BQuarterBegin
>>> ts = pd.Timestamp('2020-05-24 05:01:15')
>>> ts + BQuarterBegin()
Timestamp('2020-06-01 05:01:15')
>>> ts + BQuarterBegin(2)
Timestamp('2020-09-01 05:01:15')
>>> ts + BQuarterBegin(startingMonth=2)
Timestamp('2020-08-03 05:01:15')
>>> ts + BQuarterBegin(-1)
Timestamp('2020-03-02 05:01:15')
```

Attributes

```
base
    Returns a copy of the calling offset object with n=1 and all other attributes equal.
```

pandas.tseries.offsets.BQuarterBegin.base

```
BQuarterBegin.base
    Returns a copy of the calling offset object with n=1 and all other attributes equal.
```
Methods

```python
__call__(*args, **kwargs) Call self as a function.
rollback Roll provided date backward to next offset only if not on offset.
rollforward Roll provided date forward to next offset only if not on offset.
```

**pandas.tseries.offsets.BQuarterBegin.__call__**

```
BQuarterBegin.__call__(*args, **kwargs)
Call self as a function.
```

**pandas.tseries.offsets.BQuarterBegin.rollback**

```
BQuarterBegin.rollback()
Roll provided date backward to next offset only if not on offset.
```

Returns

- **TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

**pandas.tseries.offsets.BQuarterBegin.rollforward**

```
BQuarterBegin.rollforward()
Roll provided date forward to next offset only if not on offset.
```

Returns

- **TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.
<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
</tr>
<tr>
<td>apply_index</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>isAnchored</td>
</tr>
<tr>
<td>is_anchored</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
</tr>
<tr>
<td>is_on_offset</td>
</tr>
<tr>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
</tr>
<tr>
<td>is_year_end</td>
</tr>
<tr>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
</tr>
</tbody>
</table>

**Properties**

- `BQuarterBegin.freqstr`
- `BQuarterBegin.kwds`
- `BQuarterBegin.name`
- `BQuarterBegin.nanos`
- `BQuarterBegin.normalize`
- `BQuarterBegin.rule_code`
- `BQuarterBegin.n`
- `BQuarterBegin.startingMonth`

**pandas.tseries.offsets.BQuarterBegin.freqstr**

- `BQuarterBegin.freqstr`

**pandas.tseries.offsets.BQuarterBegin.kwds**

- `BQuarterBegin.kwds`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BQuarterBegin.name

BQuarterBegin.name

pandas.tseries.offsets.BQuarterBegin.nanos

BQuarterBegin.nanos

pandas.tseries.offsets.BQuarterBegin.normalize

BQuarterBegin.normalize

pandas.tseries.offsets.BQuarterBegin.rule_code

BQuarterBegin.rule_code

pandas.tseries.offsets.BQuarterBegin.n

BQuarterBegin.n

pandas.tseries.offsets.BQuarterBegin.startingMonth

BQuarterBegin.startingMonth
Methods

BQuarterBegin.apply

BQuarterBegin.apply_index(other)

BQuarterBegin.copy

BQuarterBegin.isAnchored

BQuarterBegin.isOnOffset

BQuarterBegin.is_anchored

BQuarterBegin.is_on_offset

BQuarterBegin.__call__(*args, **kwargs)  Call self as a function.

BQuarterBegin.is_month_start

BQuarterBegin.is_month_end

BQuarterBegin.is_quarter_start

BQuarterBegin.is_quarter_end

BQuarterBegin.is_year_start

BQuarterBegin.is_year_end

pandas.tseries.offsets.BQuarterBegin.apply

BQuarterBegin.apply()

pandas.tseries.offsets.BQuarterBegin.apply_index

BQuarterBegin.apply_index(other)

pandas.tseries.offsets.BQuarterBegin.copy

BQuarterBegin.copy()
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
pandas.tseries.offsets.BQuarterBegin.isAnchored
BQuarterBegin.isAnchored()

pandas.tseries.offsets.BQuarterBegin.onOffset
BQuarterBegin.onOffset()

pandas.tseries.offsets.BQuarterBegin.is_anchored
BQuarterBegin.is_anchored()

pandas.tseries.offsets.BQuarterBegin.is_on_offset
BQuarterBegin.is_on_offset()

pandas.tseries.offsets.BQuarterBegin.is_month_start
BQuarterBegin.is_month_start()

pandas.tseries.offsets.BQuarterBegin.is_month_end
BQuarterBegin.is_month_end()

pandas.tseries.offsets.BQuarterBegin.is_quarter_start
BQuarterBegin.is_quarter_start()

pandas.tseries.offsets.BQuarterBegin.is_quarter_end
BQuarterBegin.is_quarter_end()

pandas.tseries.offsets.BQuarterBegin.is_year_start
BQuarterBegin.is_year_start()
```
3.7.19 QuarterEnd

**QuarterEnd**

DateOffset increments between Quarter end dates.

**pandas.tseries.offsets.QuarterEnd**

*class* pandas.tseries.offsets.QuarterEnd

DateOffset increments between Quarter end dates.


**Attributes**

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.QuarterEnd.base**

QuarterEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th>kwds</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
<th>startingMonth</th>
</tr>
</thead>
</table>

**Methods**

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, *<em>kwargs)</em></th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>
pandas.tseries.offsets.QuarterEnd.__call__

QuarterEnd.__call__(*args, **kwargs)
    Call self as a function.

pandas.tseries.offsets.QuarterEnd.rollback

QuarterEnd.rollback()
    Roll provided date backward to next offset only if not on offset.

    Returns
    
    TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.QuarterEnd.rollforward

QuarterEnd.rollforward()
    Roll provided date forward to next offset only if not on offset.

    Returns
    
    TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>

Properties

- `QuarterEnd.freqstr`
- `QuarterEnd.kwds`
- `QuarterEnd.name`
- `QuarterEnd.nanos`
- `QuarterEnd.normalize`
- `QuarterEnd.rule_code`
- `QuarterEnd.n`
- `QuarterEnd.startingMonth`

---

3.7. Date offsets
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.QuarterEnd.rule_code

QuarterEnd.rule_code

pandas.tseries.offsets.QuarterEnd.n

QuarterEnd.n

pandas.tseries.offsets.QuarterEnd.startingMonth

QuarterEnd.startingMonth

Methods

QuarterEnd.apply

QuarterEnd.apply_index(other)

QuarterEnd.copy

QuarterEnd.isAnchored

QuarterEnd.onOffset

QuarterEnd.is_anchored

QuarterEnd.is_on_offset

QuarterEnd.__call__(*args, **kwargs) Call self as a function.

QuarterEnd.is_month_start

QuarterEnd.is_month_end

QuarterEnd.is_quarter_start

QuarterEnd.is_quarter_end

QuarterEnd.is_year_start

QuarterEnd.is_year_end
pandas.tseries.offsets.QuarterEnd.apply
QuarterEnd.apply()

pandas.tseries.offsets.QuarterEnd.apply_index
QuarterEnd.apply_index(other)

pandas.tseries.offsets.QuarterEnd.copy
QuarterEnd.copy()

pandas.tseries.offsets.QuarterEnd.isAnchored
QuarterEnd.isAnchored()

pandas.tseries.offsets.QuarterEnd.onOffset
QuarterEnd.onOffset()

pandas.tseries.offsets.QuarterEnd.is_anchored
QuarterEnd.is_anchored()

pandas.tseries.offsets.QuarterEnd.is_on_offset
QuarterEnd.is_on_offset()

pandas.tseries.offsets.QuarterEnd.is_month_start
QuarterEnd.is_month_start()

pandas.tseries.offsets.QuarterEnd.is_month_end
QuarterEnd.is_month_end()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.QuarterEnd.is_quarter_start

QuarterEnd.is_quarter_start()

pandas.tseries.offsets.QuarterEnd.is_quarter_end

QuarterEnd.is_quarter_end()

pandas.tseries.offsets.QuarterEnd.is_year_start

QuarterEnd.is_year_start()

pandas.tseries.offsets.QuarterEnd.is_year_end

QuarterEnd.is_year_end()

3.7.20 QuarterBegin

<table>
<thead>
<tr>
<th>QuarterBegin</th>
<th>DateOffset increments between Quarter start dates.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.QuarterBegin
class pandas.tseries.offsets.QuarterBegin

DateOffset increments between Quarter start dates.

startingMonth = 1 corresponds to dates like 1/01/2007, 4/01/2007, ... startingMonth = 2 corresponds to dates like 2/01/2007, 5/01/2007, ... startingMonth = 3 corresponds to dates like 3/01/2007, 6/01/2007, ...

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.QuarterBegin.base

QuarterBegin.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th><strong>call</strong>(*args, **kwargs)</th>
<th>Call self as a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.QuarterBegin.__call__

QuarterBegin.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.QuarterBegin.rollback

QuarterBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.QuarterBegin.rollforward

QuarterBegin.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.
<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>

**Properties**

- `QuarterBegin.freqstr`
- `QuarterBegin.kwds`
- `QuarterBegin.name`
- `QuarterBegin.nanos`
- `QuarterBegin.normalize`
- `QuarterBegin.rule_code`
- `QuarterBegin.n`
- `QuarterBegin.startingMonth`

**pandas.tseries.offsets.QuarterBegin.freqstr**

- `QuarterBegin.freqstr`

**pandas.tseries.offsets.QuarterBegin.kwds**

- `QuarterBegin.kwds`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.QuarterBegin.name

QuarterBegin.name

pandas.tseries.offsets.QuarterBegin.nanos

QuarterBegin.nanos

pandas.tseries.offsets.QuarterBegin.normalize

QuarterBegin.normalize

pandas.tseries.offsets.QuarterBegin.rule_code

QuarterBegin.rule_code

pandas.tseries.offsets.QuarterBegin.n

QuarterBegin.n

pandas.tseries.offsets.QuarterBegin.startingMonth

QuarterBegin.startingMonth

3.7. Date offsets
Methods

QuarterBegin.apply

QuarterBegin.apply_index(other)

QuarterBegin.copy

QuarterBegin.isAnchored

QuarterBegin.onOffset

QuarterBegin.is_anchored

QuarterBegin.is_on_offset

QuarterBegin.__call__(*args, **kwargs) Call self as a function.

QuarterBegin.is_month_start

QuarterBegin.is_month_end

QuarterBegin.is_quarter_start

QuarterBegin.is_quarter_end

QuarterBegin.is_year_start

QuarterBegin.is_year_end

pandas.tseries.offsets.QuarterBegin.apply

QuarterBegin.apply()

pandas.tseries.offsets.QuarterBegin.apply_index

QuarterBegin.apply_index(other)

pandas.tseries.offsets.QuarterBegin.copy

QuarterBegin.copy()
pandas.tseries.offsets.QuarterBegin.isAnchored
QuarterBegin.isAnchored()

pandas.tseries.offsets.QuarterBegin.onOffset
QuarterBegin.onOffset()

pandas.tseries.offsets.QuarterBegin.is_anchored
QuarterBegin.is_anchored()

pandas.tseries.offsets.QuarterBegin.is_on_offset
QuarterBegin.is_on_offset()

pandas.tseries.offsets.QuarterBegin.is_month_start
QuarterBegin.is_month_start()

pandas.tseries.offsets.QuarterBegin.is_month_end
QuarterBegin.is_month_end()

pandas.tseries.offsets.QuarterBegin.is_quarter_start
QuarterBegin.is_quarter_start()

pandas.tseries.offsets.QuarterBegin.is_quarter_end
QuarterBegin.is_quarter_end()

pandas.tseries.offsets.QuarterBegin.is_year_start
QuarterBegin.is_year_start()
pandas.tseries.offsets.BYearEnd

class pandas.tseries.offsets.BYearEnd
    DateOffset increments between the last business day of the year.

Examples

```python
>>> from pandas.tseries.offsets import BYearEnd
>>> ts = pd.Timestamp('2020-05-24 05:01:15')
>>> ts - BYearEnd()
Timestamp('2019-12-31 05:01:15')
>>> ts + BYearEnd()
Timestamp('2020-12-31 05:01:15')
>>> ts + BYearEnd(3)
Timestamp('2022-12-30 05:01:15')
>>> ts + BYearEnd(-3)
Timestamp('2017-12-29 05:01:15')
>>> ts + BYearEnd(month=11)
Timestamp('2020-11-30 05:01:15')
```

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BYearEnd.base

BYearEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th>freqstr</th>
<th>kwds</th>
<th>month</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
</tr>
</thead>
</table>

__call__(*args, **kwargs)  
Call self as a function.

rollback  
Roll provided date backward to next offset only if not on offset.

rollforward  
Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BYearEnd.__call__

BYearEnd.__call__(*args, **kwargs)

Call self as a function.

pandas.tseries.offsets.BYearEnd.rollback

BYearEnd.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

DataFrame  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.BYearEnd.rollforward

BYearEnd.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

DataFrame  Rolled timestamp if not on offset, otherwise unchanged timestamp.
apply
apply_index
copy
isAnchored
is_anchored
is_month_end
is_month_start
is_on_offset
is_quarter_end
is_quarter_start
is_year_end
is_year_start
onOffset

Properties

BYearEnd.freqstr

BYearEnd.kwds

BYearEnd.name

BYearEnd.nanos

BYearEnd.normalize

BYearEnd.rule_code

BYearEnd.n

BYearEnd.month

pandas.tseries.offsets.BYearEnd.freqstr

BYearEnd.freqstr

pandas.tseries.offsets.BYearEnd.kwds

BYearEnd.kwds
3.7. Date offsets

pandas.tseries.offsets.BYearEnd.name

BYearEnd.name

pandas.tseries.offsets.BYearEnd.nanos

BYearEnd.nanos

pandas.tseries.offsets.BYearEnd.normalize

BYearEnd.normalize

pandas.tseries.offsets.BYearEnd.rule_code

BYearEnd.rule_code

pandas.tseries.offsets.BYearEnd.n

BYearEnd.n

pandas.tseries.offsets.BYearEnd.month

BYearEnd.month
Methods

- `BYearEnd.apply`
- `BYearEnd.apply_index(other)`
- `BYearEnd.copy`
- `BYearEnd.isAnchored`
- `BYearEnd.onOffset`
- `BYearEnd.is_anchored`
- `BYearEnd.is_on_offset`
- `BYearEnd.__call__(*args, **kwargs)` Call self as a function.
- `BYearEnd.is_month_start`
- `BYearEnd.is_month_end`
- `BYearEnd.is_quarter_start`
- `BYearEnd.is_quarter_end`
- `BYearEnd.is_year_start`
- `BYearEnd.is_year_end`

```python
pandas.tseries.offsets.BYearEnd.apply
BYearEnd.apply()

pandas.tseries.offsets.BYearEnd.apply_index
BYearEnd.apply_index(other)

pandas.tseries.offsets.BYearEnd.copy
BYearEnd.copy()
```
pandas.tseries.offsets.BYearEnd.isAnchored

BYearEnd.isAnchored()

pandas.tseries.offsets.BYearEnd.onOffset

BYearEnd.onOffset()

pandas.tseries.offsets.BYearEnd.is_anchored

BYearEnd.is_anchored()

pandas.tseries.offsets.BYearEnd.is_on_offset

BYearEnd.is_on_offset()

pandas.tseries.offsets.BYearEnd.is_month_start

BYearEnd.is_month_start()

pandas.tseries.offsets.BYearEnd.is_month_end

BYearEnd.is_month_end()

pandas.tseries.offsets.BYearEnd.is_quarter_start

BYearEnd.is_quarter_start()

pandas.tseries.offsets.BYearEnd.is_quarter_end

BYearEnd.is_quarter_end()

pandas.tseries.offsets.BYearEnd.is_year_start

BYearEnd.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BYearEnd.is_year_end

BYearEnd.is_year_end()

3.7.22 BYearBegin

<table>
<thead>
<tr>
<th>BYearBegin</th>
<th>DateOffset increments between the first business day of the year.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.BYearBegin

class pandas.tseries.offsets.BYearBegin

DateOffset increments between the first business day of the year.

Examples

```python
>>> from pandas.tseries.offsets import BYearBegin
>>> ts = pd.Timestamp('2020-05-24 05:01:15')
>>> ts + BYearBegin()
Timestamp('2021-01-01 05:01:15')
>>> ts - BYearBegin()
Timestamp('2020-01-01 05:01:15')
>>> ts + BYearBegin(-1)
Timestamp('2020-01-01 05:01:15')
>>> ts + BYearBegin(2)
Timestamp('2022-01-03 05:01:15')
```

Attributes

| base | Returns a copy of the calling offset object with n=1 and all other attributes equal. |

pandas.tseries.offsets.BYearBegin.base

BYearBegin.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>freqstr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kwds</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>nanos</td>
<td></td>
</tr>
<tr>
<td>normalize</td>
<td></td>
</tr>
<tr>
<td>rule_code</td>
<td></td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.BYearBegin.__call__

BYearBegin.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.BYearBegin.rollback

BYearBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.BYearBegin.rollforward

BYearBegin.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
</tr>
<tr>
<td>apply_index</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>isAnchored</td>
</tr>
<tr>
<td>is_anchored</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
</tr>
<tr>
<td>is_on_offset</td>
</tr>
<tr>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
</tr>
<tr>
<td>is_year_end</td>
</tr>
<tr>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
</tr>
</tbody>
</table>
Properties

BYearBegin.freqstr
BYearBegin.kwds
BYearBegin.name
BYearBegin.nanos
BYearBegin.normalize
BYearBegin.rule_code
BYearBegin.n
BYearBegin.month

pandas.tseries.offsets.BYearBegin.freqstr
BYearBegin.freqstr

pandas.tseries.offsets.BYearBegin.kwds
BYearBegin.kwds

pandas.tseries.offsets.BYearBegin.name
BYearBegin.name

pandas.tseries.offsets.BYearBegin.nanos
BYearBegin.nanos

pandas.tseries.offsets.BYearBegin.normalize
BYearBegin.normalize
**pandas.tseries.offsets.BYearBegin.rule_code**

`BYearBegin.rule_code`

**pandas.tseries.offsets.BYearBegin.n**

`BYearBegin.n`

**pandas.tseries.offsets.BYearBegin.month**

`BYearBegin.month`

**Methods**

- `BYearBegin.apply`
- `BYearBegin.apply_index(other)`
- `BYearBegin.copy`
- `BYearBegin.isAnchored`
- `BYearBegin.onOffset`
- `BYearBegin.is_anchored`
- `BYearBegin.is_on_offset`
- `BYearBegin.__call__(*args, **kwargs)` Call self as a function.
- `BYearBegin.is_month_start`
- `BYearBegin.is_month_end`
- `BYearBegin.is_quarter_start`
- `BYearBegin.is_quarter_end`
- `BYearBegin.is_year_start`
- `BYearBegin.is_year_end`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.BYearBegin.apply
BYearBegin.apply()

pandas.tseries.offsets.BYearBegin.apply_index
BYearBegin.apply_index(other)

pandas.tseries.offsets.BYearBegin.copy
BYearBegin.copy()

pandas.tseries.offsets.BYearBegin.isAnchored
BYearBegin.isAnchored()

pandas.tseries.offsets.BYearBegin.onOffset
BYearBegin.onOffset()

pandas.tseries.offsets.BYearBegin.is_anchored
BYearBegin.is_anchored()

pandas.tseries.offsets.BYearBegin.is_on_offset
BYearBegin.is_on_offset()

pandas.tseries.offsets.BYearBegin.is_month_start
BYearBegin.is_month_start()

pandas.tseries.offsets.BYearBegin.is_month_end
BYearBegin.is_month_end()
BYearBegin.is_quarter_start()

BYearBegin.is_quarter_end()

BYearBegin.is_year_start()

BYearBegin.is_year_end()

### 3.7.23 YearEnd

<table>
<thead>
<tr>
<th>YearEnd</th>
<th>DateOffset increments between calendar year ends.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.YearEnd**

**class** pandas.tseries.offsets.YearEnd

DateOffset increments between calendar year ends.

**Attributes**

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

**pandas.tseries.offsets.YearEnd.base**

YearEnd.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

`pandas.tseries.offsets.YearEnd.__call__`

YearEnd.__call__(*args, **kwargs*)
Call self as a function.

`pandas.tseries.offsets.YearEnd.rollback`

YearEnd.rollback()
Roll provided date backward to next offset only if not on offset.

Returns
- **TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

`pandas.tseries.offsets.YearEnd.rollforward`

YearEnd.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns
- **TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>apply_index</td>
<td></td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>isAnchored</td>
<td></td>
</tr>
<tr>
<td>is_anchored</td>
<td></td>
</tr>
<tr>
<td>is_month_end</td>
<td></td>
</tr>
<tr>
<td>is_month_start</td>
<td></td>
</tr>
<tr>
<td>is_on_offset</td>
<td></td>
</tr>
<tr>
<td>is_quarter_end</td>
<td></td>
</tr>
<tr>
<td>is_quarter_start</td>
<td></td>
</tr>
<tr>
<td>is_year_end</td>
<td></td>
</tr>
<tr>
<td>is_year_start</td>
<td></td>
</tr>
<tr>
<td>onOffset</td>
<td></td>
</tr>
</tbody>
</table>
Properties

YearEnd.freqstr
YearEnd.kwds
YearEnd.name
YearEnd.nanos
YearEnd.normalize
YearEnd.rule_code
YearEnd.n
YearEnd.month

pandas.tseries.offsets.YearEnd.freqstr
YearEnd.freqstr

pandas.tseries.offsets.YearEnd.kwds
YearEnd.kwds

pandas.tseries.offsets.YearEnd.name
YearEnd.name

pandas.tseries.offsets.YearEnd.nanos
YearEnd.nanos

pandas.tseries.offsets.YearEnd.normalize
YearEnd.normalize
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.YearEnd.rule_code

YearEnd.rule_code

pandas.tseries.offsets.YearEnd.n

YearEnd.n

pandas.tseries.offsets.YearEnd.month

YearEnd.month

Methods

YearEnd.apply

YearEnd.apply_index(other)

YearEnd.copy

YearEnd.isAnchored

YearEnd.onOffset

YearEnd.is_anchored

YearEnd.is_on_offset

YearEnd.__call__(*args, **kwargs) Call self as a function.

YearEnd.is_month_start

YearEnd.is_month_end

YearEnd.is_quarter_start

YearEnd.is_quarter_end

YearEnd.is_year_start

YearEnd.is_year_end
pandas.tseries.offsets.YearEnd.apply
YearEnd.apply()

pandas.tseries.offsets.YearEnd.apply_index
YearEnd.apply_index(other)

pandas.tseries.offsets.YearEnd.copy
YearEnd.copy()

pandas.tseries.offsets.YearEnd.isAnchored
YearEnd.isAnchored()

pandas.tseries.offsets.YearEnd.onOffset
YearEnd.onOffset()

pandas.tseries.offsets.YearEnd.is_anchored
YearEnd.is_anchored()

pandas.tseries.offsets.YearEnd.is_on_offset
YearEnd.is_on_offset()

pandas.tseries.offsets.YearEnd.is_month_start
YearEnd.is_month_start()

pandas.tseries.offsets.YearEnd.is_month_end
YearEnd.is_month_end()
pandas.tseries.offsets.YearEnd.is_quarter_start

YearEnd.is_quarter_start()

pandas.tseries.offsets.YearEnd.is_quarter_end

YearEnd.is_quarter_end()

pandas.tseries.offsets.YearEnd.is_year_start

YearEnd.is_year_start()

pandas.tseries.offsets.YearEnd.is_year_end

YearEnd.is_year_end()

3.7.24 YearBegin

<table>
<thead>
<tr>
<th>YearBegin</th>
<th>DateOffset increments between calendar year begin dates.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.YearBegin

class pandas.tseries.offsets.YearBegin

DateOffset increments between calendar year begin dates.

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.YearBegin.base

YearBegin.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

_____call__(*args, **kwargs) Call self as a function.

rollback Roll provided date backward to next offset only if not on offset.

rollforward Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.YearBegin.__call__

YearBegin.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.YearBegin.rollback

YearBegin.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.YearBegin.rollforward

YearBegin.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>
Properties

YearBegin.freqstr

YearBegin.kwds

YearBegin.name

YearBegin.nanos

YearBegin.normalize

YearBegin.rule_code

YearBegin.n

YearBegin.month

pandas.tseries.offsets.YearBegin.freqstr

YearBegin.freqstr

pandas.tseries.offsets.YearBegin.kwds

YearBegin.kwds

pandas.tseries.offsets.YearBegin.name

YearBegin.name

pandas.tseries.offsets.YearBegin.nanos

YearBegin.nanos

pandas.tseries.offsets.YearBegin.normalize

YearBegin.normalize
pandas.tseries.offsets.YearBegin.rule_code

YearBegin.rule_code

pandas.tseries.offsets.YearBegin.n

YearBegin.n

pandas.tseries.offsets.YearBegin.month

YearBegin.month

Methods

YearBegin.apply

YearBegin.apply_index(other)

YearBegin.copy

YearBegin.isAnchored

YearBegin.onOffset

YearBegin.is_anchored

YearBegin.is_on_offset

YearBegin.__call__(*args, **kwargs) Call self as a function.

YearBegin.is_month_start

YearBegin.is_month_end

YearBegin.is_quarter_start

YearBegin.is_quarter_end

YearBegin.is_year_start

YearBegin.is_year_end
pandas.tseries.offsets.YearBegin.apply

YearBegin.apply()

pandas.tseries.offsets.YearBegin.apply_index

YearBegin.apply_index(other)

pandas.tseries.offsets.YearBegin.copy

YearBegin.copy()

pandas.tseries.offsets.YearBegin.isAnchored

YearBegin.isAnchored()

pandas.tseries.offsets.YearBegin.onOffset

YearBegin.onOffset()

pandas.tseries.offsets.YearBegin.is_anchored

YearBegin.is_anchored()

pandas.tseries.offsets.YearBegin.is_on_offset

YearBegin.is_on_offset()

pandas.tseries.offsets.YearBegin.is_month_start

YearBegin.is_month_start()

pandas.tseries.offsets.YearBegin.is_month_end

YearBegin.is_month_end()
3.7.25 FY5253

FY5253 Describes 52-53 week fiscal year.

pandas.tseries.offsets.FY5253
class pandas.tseries.offsets.FY5253

Describes 52-53 week fiscal year. This is also known as a 4-4-5 calendar.
It is used by companies that desire that their fiscal year always end on the same day of the week.
It is a method of managing accounting periods. It is a common calendar structure for some industries, such as retail, manufacturing and parking industry.

For more information see: https://en.wikipedia.org/wiki/4-4-5_calendar

The year may either:
• end on the last X day of the Y month.
• end on the last X day closest to the last day of the Y month.

X is a specific day of the week. Y is a certain month of the year

Parameters

n [int]

weekday [int {0, 1, ..., 6}, default 0] A specific integer for the day of the week.
• 0 is Monday
• 1 is Tuesday
• 2 is Wednesday
• 3 is Thursday
• 4 is Friday
• 5 is Saturday
• 6 is Sunday.

startingMonth [int {1, 2, …, 12}, default 1] The month in which the fiscal year ends.

variation [str, default “nearest”] Method of employing 4-4-5 calendar.

There are two options:
  • “nearest” means year end is weekday closest to last day of month in year.
  • “last” means year end is final weekday of the final month in fiscal year.

Attributes

<table>
<thead>
<tr>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a copy of the calling offset object with n=1 and all other attributes equal.</td>
</tr>
</tbody>
</table>

pandas.tseries.offsets.FY5253.base

FY5253 .base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr
kwds
n
name
nanos
normalize
rule_code
startingMonth
variation
weekday

Methods

__call__(*args, **kwargs) Call self as a function.

rollback Roll provided date backward to next offset only if not on offset.

rollforward Roll provided date forward to next offset only if not on offset.
pandas.tseries.offsets.FY5253.__call__

FY5253.__call__(*args, **kwargs)
Call self as a function.

pandas.tseries.offsets.FY5253.rollback

FY5253.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.FY5253.rollforward

FY5253.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply_index</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>get_rule_code_suffix</td>
</tr>
<tr>
<td>get_year_end</td>
</tr>
<tr>
<td>isAnchored</td>
</tr>
<tr>
<td>is_anchored</td>
</tr>
<tr>
<td>is_month_end</td>
</tr>
<tr>
<td>is_month_start</td>
</tr>
<tr>
<td>is_on_offset</td>
</tr>
<tr>
<td>is_quarter_end</td>
</tr>
<tr>
<td>is_quarter_start</td>
</tr>
<tr>
<td>is_year_end</td>
</tr>
<tr>
<td>is_year_start</td>
</tr>
<tr>
<td>onOffset</td>
</tr>
</tbody>
</table>
Properties

FY5253.freqstr
FY5253.kwds
FY5253.name
FY5253.nanos
FY5253.normalize
FY5253.rule_code
FY5253.n
FY5253.startingMonth
FY5253.variation
FY5253.weekday

pandas.tseries.offsets.FY5253.freqstr
FY5253.freqstr

pandas.tseries.offsets.FY5253.kwds
FY5253.kwds

pandas.tseries.offsets.FY5253.name
FY5253.name

pandas.tseries.offsets.FY5253.nanos
FY5253.nanos
pandas.tseries.offsets.FY5253.normalize

FY5253.normalize

pandas.tseries.offsets.FY5253.rule_code

FY5253.rule_code

pandas.tseries.offsets.FY5253.n

FY5253.n

pandas.tseries.offsets.FY5253.startingMonth

FY5253.startingMonth

pandas.tseries.offsets.FY5253.variation

FY5253.variation

pandas.tseries.offsets.FY5253.weekday

FY5253.weekday
Methods

FY5253.apply
FY5253.apply_index(other)
FY5253.copy
FY5253.get_rule_code_suffix
FY5253.get_year_end
FY5253.isAnchored
FY5253.onOffset
FY5253.is_anchored
FY5253.is_on_offset
FY5253.__call__(*args, **kwargs) Call self as a function.
FY5253.is_month_start
FY5253.is_month_end
FY5253.is_quarter_start
FY5253.is_quarter_end
FY5253.is_year_start
FY5253.is_year_end
3.7. Date offsets

pandas.tseries.offsets.FY5253.apply_index

FY5253.apply_index(other)

pandas.tseries.offsets.FY5253.copy

FY5253.copy()

pandas.tseries.offsets.FY5253.get_rule_code_suffix

FY5253.get_rule_code_suffix()

pandas.tseries.offsets.FY5253.get_year_end

FY5253.get_year_end()

pandas.tseries.offsets.FY5253.isAnchored

FY5253.isAnchored()

pandas.tseries.offsets.FY5253.onOffset

FY5253.onOffset()

pandas.tseries.offsets.FY5253.is_anchored

FY5253.is_anchored()

pandas.tseries.offsets.FY5253.is_on_offset

FY5253.is_on_offset()

pandas.tseries.offsets.FY5253.is_month_start

FY5253.is_month_start()
**3.7.26 FY5253Quarter**

<table>
<thead>
<tr>
<th>FY5253Quarter</th>
<th>DateOffset increments between business quarter dates for 52-53 week fiscal year (also known as a 4-4-5 calendar).</th>
</tr>
</thead>
</table>

**class pandas.tseries.offsets.FY5253Quarter**

DateOffset increments between business quarter dates for 52-53 week fiscal year (also known as a 4-4-5 calendar).

It is used by companies that desire that their fiscal year always end on the same day of the week.

It is a method of managing accounting periods. It is a common calendar structure for some industries, such as retail, manufacturing and parking industry.

For more information see: https://en.wikipedia.org/wiki/4-4-5_calendar

The year may either:

• end on the last X day of the Y month.
• end on the last X day closest to the last day of the Y month.

X is a specific day of the week. Y is a certain month of the year

startingMonth = 1 corresponds to dates like 1/31/2007, 4/30/2007, ... startingMonth = 2 corresponds to dates like 2/28/2007, 5/31/2007, ... startingMonth = 3 corresponds to dates like 3/30/2007, 6/29/2007, ...

**Parameters**

n [int]
**weekday** [int \{0, 1, \ldots, 6\}, default 0] A specific integer for the day of the week.

- 0 is Monday
- 1 is Tuesday
- 2 is Wednesday
- 3 is Thursday
- 4 is Friday
- 5 is Saturday
- 6 is Sunday.

**startingMonth** [int \{1, 2, \ldots, 12\}, default 1] The month in which fiscal years end.

**qtr_with_extra_week** [int \{1, 2, 3, 4\}, default 1] The quarter number that has the leap or 14 week when needed.

**variation** [str, default “nearest”] Method of employing 4-4-5 calendar.

There are two options:

- “nearest” means year end is **weekday** closest to last day of month in year.
- “last” means year end is final **weekday** of the final month in fiscal year.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base</code></td>
<td>Returns a copy of the calling offset object with n=1 and all other attributes equal.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.FY5253Quarter.base**

`FY5253Quarter.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>freqstr</code></td>
</tr>
<tr>
<td><code>kwds</code></td>
</tr>
<tr>
<td><code>n</code></td>
</tr>
<tr>
<td><code>name</code></td>
</tr>
<tr>
<td><code>nanos</code></td>
</tr>
<tr>
<td><code>normalize</code></td>
</tr>
<tr>
<td><code>qtr_with_extra_week</code></td>
</tr>
<tr>
<td><code>rule_code</code></td>
</tr>
<tr>
<td><code>startingMonth</code></td>
</tr>
<tr>
<td><code>variation</code></td>
</tr>
<tr>
<td><code>weekday</code></td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__call__(*args, **kwargs)*</code></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td><code>rollback</code></td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td><code>rollforward</code></td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

`pandas.tseries.offsets.FY5253Quarter.__call__`

FY5253Quarter.__call__(*args, **kwargs)*

Call self as a function.

`pandas.tseries.offsets.FY5253Quarter.rollback`

FY5253Quarter.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

`TimeStamp` Rolled timestamp if not on offset, otherwise unchanged timestamp.

`pandas.tseries.offsets.FY5253Quarter.rollforward`

FY5253Quarter.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

`TimeStamp` Rolled timestamp if not on offset, otherwise unchanged timestamp.

```python
apply
apply_index
copy
get_rule_code_suffix
get_weeks
isAnchored
is_anchored
is_month_end
is_month_start
is_on_offset
is_quarter_end
is_quarter_start
is_year_end
is_year_start
onOffset
year_has_extra_week
```
### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FY5253Quarter.freqstr</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.kwds</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.name</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.nanos</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.normalize</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.rule_code</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.n</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.qtr_with_extra_week</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.startingMonth</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.variation</code></td>
<td></td>
</tr>
<tr>
<td><code>FY5253Quarter.weekday</code></td>
<td></td>
</tr>
</tbody>
</table>

### Date offsets

- `pandas.tseries.offsets.FY5253Quarter.freqstr`
- `FY5253Quarter.freqstr`
- `pandas.tseries.offsets.FY5253Quarter.kwds`
- `FY5253Quarter.kwds`
- `pandas.tseries.offsets.FY5253Quarter.name`
- `FY5253Quarter.name`
- `pandas.tseries.offsets.FY5253Quarter.nanos`
- `FY5253Quarter.nanos`
pandas.tseries.offsets.FY5253Quarter.normalize
FY5253Quarter.normalize

pandas.tseries.offsets.FY5253Quarter.rule_code
FY5253Quarter.rule_code

pandas.tseries.offsets.FY5253Quarter.n
FY5253Quarter.n

pandas.tseries.offsets.FY5253Quarter.qtr_with_extra_week
FY5253Quarter.qtr_with_extra_week

pandas.tseries.offsets.FY5253Quarter.startingMonth
FY5253Quarter.startingMonth

pandas.tseries.offsets.FY5253Quarter.variation
FY5253Quarter.variation

pandas.tseries.offsets.FY5253Quarter.weekday
FY5253Quarter.weekday
Methods

FY5253Quarter.apply

FY5253Quarter.apply_index(other)

FY5253Quarter.copy

FY5253Quarter.get_rule_code_suffix

FY5253Quarter.get_weeks

FY5253Quarter.isAnchored

FY5253Quarter.onOffset

FY5253Quarter.is_anchored

FY5253Quarter.is_on_offset

FY5253Quarter.year_has_extra_week

FY5253Quarter.__call__(*args, **kwargs)  # Call self as a function.

FY5253Quarter.is_month_start

FY5253Quarter.is_month_end

FY5253Quarter.is_quarter_start

FY5253Quarter.is_quarter_end

FY5253Quarter.is_year_start

FY5253Quarter.is_year_end

pandas.tseries.offsets.FY5253Quarter.apply

FY5253Quarter.apply()

pandas.tseries.offsets.FY5253Quarter.apply_index

FY5253Quarter.apply_index(other)
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.FY5253Quarter.copy
FY5253Quarter.copy()

pandas.tseries.offsets.FY5253Quarter.get_rule_code_suffix
FY5253Quarter.get_rule_code_suffix()

pandas.tseries.offsets.FY5253Quarter.get_weeks
FY5253Quarter.get_weeks()

pandas.tseries.offsets.FY5253Quarter.isAnchored
FY5253Quarter.isAnchored()

pandas.tseries.offsets.FY5253Quarter.onOffset
FY5253Quarter.onOffset()

pandas.tseries.offsets.FY5253Quarter.is_anchored
FY5253Quarter.is_anchored()

pandas.tseries.offsets.FY5253Quarter.is_on_offset
FY5253Quarter.is_on_offset()

pandas.tseries.offsets.FY5253Quarter.year_has_extra_week
FY5253Quarter.year_has_extra_week()

pandas.tseries.offsets.FY5253Quarter.is_month_start
FY5253Quarter.is_month_start()
pandas.tseries.offsets.FY5253Quarter.is_month_end

FY5253Quarter.is_month_end()

pandas.tseries.offsets.FY5253Quarter.is_quarter_start

FY5253Quarter.is_quarter_start()

pandas.tseries.offsets.FY5253Quarter.is_quarter_end

FY5253Quarter.is_quarter_end()

pandas.tseries.offsets.FY5253Quarter.is_year_start

FY5253Quarter.is_year_start()

pandas.tseries.offsets.FY5253Quarter.is_year_end

FY5253Quarter.is_year_end()

3.7.27 Easter

<table>
<thead>
<tr>
<th>Easter</th>
<th>DateOffset for the Easter holiday using logic defined in dateutil.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.Easter

class pandas.tseries.offsets.Easter

DateOffset for the Easter holiday using logic defined in dateutil.

Right now uses the revised method which is valid in years 1583-4099.

Attributes

| base | Returns a copy of the calling offset object with n=1 and all other attributes equal. |
pandas.tseries.offsets.Easter.base

Easter.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr
kwds
n
name
nanos
normalize
rule_code

Methods

__call__(args, **kwargs)  Call self as a function.
rollback
Roll provided date backward to next offset only if not on offset.
rollforward
Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Easter.__call__

Easter.__call__(args, **kwargs)
Call self as a function.

pandas.tseries.offsets.Easter.rollback

Easter.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Easter.rollforward

Easter.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
Properties

```
Easter.freqstr
Easter.kwds
Easter.name
Easter.nanos
Easter.normalize
Easter.rule_code
Easter.n
```

```
pandas.tseries.offsets.Easter.freqstr
```

```
Easter.freqstr
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Easter.kwds

Easter.kwds

pandas.tseries.offsets.Easter.name

Easter.name

pandas.tseries.offsets.Easter.nanos

Easter.nanos

pandas.tseries.offsets.Easter.normalize

Easter.normalize

pandas.tseries.offsets.Easter.rule_code

Easter.rule_code

pandas.tseries.offsets.Easter.n

Easter.n
Methods

Easter.apply

Easter.apply_index(other)

Easter.copy

Easter.isAnchored

Easter.onOffset

Easter.is_anchored

Easter.is_on_offset

Easter.__call__(*args, **kwargs) Call self as a function.

Easter.is_month_start

Easter.is_month_end

Easter.is_quarter_start

Easter.is_quarter_end

Easter.is_year_start

Easter.is_year_end

pandas.tseries.offsets.Easter.apply

Easter.apply()

pandas.tseries.offsets.Easter.apply_index

Easter.apply_index(other)

pandas.tseries.offsets.Easter.copy

Easter.copy()
```
pandas.tseries.offsets.Easter.isAnchored
Easter.isAnchored()

doctor.isAnchored()

pandas.tseries.offsets.Easter.onOffset
Easter.onOffset()

pandas.tseries.offsets.Easter.is_anchored
Easter.is_anchored()

pandas.tseries.offsets.Easter.is_on_offset
Easter.is_on_offset()

pandas.tseries.offsets.Easter.is_month_start
Easter.is_month_start()

pandas.tseries.offsets.Easter.is_month_end
Easter.is_month_end()

pandas.tseries.offsets.Easter.is_quarter_start
Easter.is_quarter_start()

pandas.tseries.offsets.Easter.is_quarter_end
Easter.is_quarter_end()

pandas.tseries.offsets.Easter.is_year_start
Easter.is_year_start()
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
pandas.tseries.offsets.Easter.is_year_end

Easter.is_year_end()
```

### 3.7.28 Tick

#### Attributes

```python
class pandas.tseries.offsets.Tick:

Attributes

- base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
```

```python
pandas.tseries.offsets.Tick.base

Tick.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
```

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta</td>
</tr>
<tr>
<td>freqstr</td>
</tr>
<tr>
<td>kwds</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>nanos</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>rule_code</td>
</tr>
</tbody>
</table>

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args,**kwags)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

3.7. Date offsets
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Tick.__call__

Tick.__call__(*args, **kwargs)

Call self as a function.

pandas.tseries.offsets.Tick.rollback

Tick.rollback()

Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Tick.rollforward

Tick.rollforward()

Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>
Properties

- `pandas.tseries.offsets.Tick.delta`
- `pandas.tseries.offsets.Tick.freqstr`
- `pandas.tseries.offsets.Tick.kwds`
- `pandas.tseries.offsets.Tick.name`
- `pandas.tseries.offsets.Tick.nanos`
- `pandas.tseries.offsets.Tick.normalize`
- `pandas.tseries.offsets.Tick.rule_code`
- `pandas.tseries.offsets.Tick.n`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Tick.normalize

Tick.normalize

pandas.tseries.offsets.Tick.rule_code

Tick.rule_code

pandas.tseries.offsets.Tick.n

Tick.n

Methods

Tick.copy

Tick.isAnchored

Tick.onOffset

Tick.is_anchored

Tick.is_on_offset

Tick.__call__(*args, **kwargs) Call self as a function.

Tick.apply

Tick.apply_index(other)

Tick.is_month_start

Tick.is_month_end

Tick.is_quarter_start

Tick.is_quarter_end

Tick.is_year_start

Tick.is_year_end
pandas.tseries.offsets.Tick.copy

Tick.copy()

pandas.tseries.offsets.Tick.isAnchored

Tick.isAnchored()

pandas.tseries.offsets.Tick.onOffset

Tick.onOffset()

pandas.tseries.offsets.Tick.is_anchored

Tick.is_anchored()

pandas.tseries.offsets.Tick.is_on_offset

Tick.is_on_offset()

pandas.tseries.offsets.Tick.apply

Tick.apply()

pandas.tseries.offsets.Tick.apply_index

Tick.apply_index(other)

pandas.tseries.offsets.Tick.is_month_start

Tick.is_month_start()

pandas.tseries.offsets.Tick.is_month_end

Tick.is_month_end()
pandas: powerful Python data analysis toolkit, Release 1.4.2

`pandas.tseries.offsets.Tick.is_quarter_start`

`Tick.is_quarter_start()`

`pandas.tseries.offsets.Tick.is_quarter_end`

`Tick.is_quarter_end()`

`pandas.tseries.offsets.Tick.is_year_start`

`Tick.is_year_start()`

`pandas.tseries.offsets.Tick.is_year_end`

`Tick.is_year_end()`

### 3.7.29 Day

![Day class](class_diagram)

#### Attributes

**Attributes**

- **base**

  Returns a copy of the calling offset object with `n=1` and all other attributes equal.

- **pandas.tseries.offsets.Day.base**

  **Day.base**

  Returns a copy of the calling offset object with `n=1` and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

```
pandas.tseries.offsets.Day.__call__

Day.__call__(*args, **kwargs*)
Call self as a function.
```

```
pandas.tseries.offsets.Day.rollback

Day.rollback()
Roll provided date backward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
```

```
pandas.tseries.offsets.Day.rollforward

Day.rollforward()
Roll provided date forward to next offset only if not on offset.

Returns

TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
```
apply
apply_index
copy
isAnchored
is_anchored
is_month_end
is_month_start
is_on_offset
is_quarter_end
is_quarter_start
is_year_end
is_year_start
onOffset

Properties

Day.delta

Day.freqstr

Day.kwds

Day.name

Day.nanos

Day.normalize

Day.rule_code

Day.n

pandas.tseries.offsets.Day.delta

Day.delta

pandas.tseries.offsets.Day.freqstr

Day.freqstr
pandas.tseries.offsets.Day.kwds

Day.kwds

pandas.tseries.offsets.Day.name

Day.name

pandas.tseries.offsets.Day.nanos

Day.nanos

pandas.tseries.offsets.Day.normalize

Day.normalize

pandas.tseries.offsets.Day.rule_code

Day.rule_code

pandas.tseries.offsets.Day.n

Day.n
Methods

- `Day.copy`
- `Day.isAnchored`
- `Day.onOffset`
- `Day.is_anchored`
- `Day.is_on_offset`
- `Day.__call__(*args, **kwargs)` Call self as a function.
- `Day.apply`
- `Day.apply_index(other)`
- `Day.is_month_start`
- `Day.is_month_end`
- `Day.is_quarter_start`
- `Day.is_quarter_end`
- `Day.is_year_start`
- `Day.is_year_end`

`pandas.tseries.offsets.Day.copy`

`Day.copy()`

`pandas.tseries.offsets.Day.isAnchored`

`Day.isAnchored()`

`pandas.tseries.offsets.Day.onOffset`

`Day.onOffset()`
pandas.tseries.offsets.Day.is_anchored

Day.is_anchored()

pandas.tseries.offsets.Day.is_on_offset

Day.is_on_offset()

pandas.tseries.offsets.Day.apply

Day.apply()

pandas.tseries.offsets.Day.apply_index

Day.apply_index(other)

pandas.tseries.offsets.Day.is_month_start

Day.is_month_start()

pandas.tseries.offsets.Day.is_month_end

Day.is_month_end()

pandas.tseries.offsets.Day.is_quarter_start

Day.is_quarter_start()

pandas.tseries.offsets.Day.is_quarter_end

Day.is_quarter_end()

pandas.tseries.offsets.Day.is_year_start

Day.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Day.is_year_end

Day.is_year_end()

### 3.7.30 Hour

Hour

#### Attributes

```
pandas.tseries.offsets.Hour
class pandas.tseries.offsets.Hour

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.Hour.base

Hour.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>delta</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>freqstr</td>
<td></td>
</tr>
<tr>
<td>kwds</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>nanos</td>
<td></td>
</tr>
<tr>
<td>normalize</td>
<td></td>
</tr>
<tr>
<td>rule_code</td>
<td></td>
</tr>
</tbody>
</table>

#### Methods

```
__call__(*args, **kwargs) Call self as a function.
rollback Roll provided date backward to next offset only if not on offset.
rollforward Roll provided date forward to next offset only if not on offset.
```
pandas.tseries.offsets.Hour.__call__

Hour.__call__(*args, **kwargs)
    Call self as a function.

pandas.tseries.offsets.Hour.rollback

Hour.rollback()
    Roll provided date backward to next offset only if not on offset.
    Returns
        TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Hour.rollforward

Hour.rollforward()
    Roll provided date forward to next offset only if not on offset.
    Returns
        TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>

## Properties

- `Hour.delta`
- `Hour.freqstr`
- `Hour.kwds`
- `Hour.name`
- `Hour.nanos`
- `Hour.normalize`
- `Hour.rule_code`
- `Hour.n`

```python
pandas.tseries.offsets.Hour.delta
```

*Hour.delta*

```python
pandas.tseries.offsets.Hour.freqstr
```

*Hour.freqstr*

```python
pandas.tseries.offsets.Hour.kwds
```

*Hour.kwds*

```python
pandas.tseries.offsets.Hour.name
```

*Hour.name*

```python
pandas.tseries.offsets.Hour.nanos
```

*Hour.nanos*
pandas.tseries.offsets.Hour.normalize

Hour.normalize

pandas.tseries.offsets.Hour.rule_code

Hour.rule_code

pandas.tseries.offsets.Hour.n

Hour.n

Methods

Hour.copy

Hour.isAnchored

Hour.onOffset

Hour.is_anchored

Hour.is_on_offset

Hour.__call__(*args, **kwargs) Call self as a function.

Hour.apply

Hour.apply_index(other)

Hour.is_month_start

Hour.is_month_end

Hour.is_quarter_start

Hour.is_quarter_end

Hour.is_year_start

Hour.is_year_end
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Hour.copy

Hour.copy()

pandas.tseries.offsets.Hour.isAnchored

Hour.isAnchored()

pandas.tseries.offsets.Hour.onOffset

Hour.onOffset()

pandas.tseries.offsets.Hour.is_anchored

Hour.is_anchored()

pandas.tseries.offsets.Hour.is_on_offset

Hour.is_on_offset()

pandas.tseries.offsets.Hour.apply

Hour.apply()

pandas.tseries.offsets.Hour.apply_index

Hour.apply_index(other)

pandas.tseries.offsets.Hour.is_month_start

Hour.is_month_start()

pandas.tseries.offsets.Hour.is_month_end

Hour.is_month_end()
pandas.tseries.offsets.Hour.is_quarter_start

Hour.is_quarter_start()

pandas.tseries.offsets.Hour.is_quarter_end

Hour.is_quarter_end()

pandas.tseries.offsets.Hour.is_year_start

Hour.is_year_start()

pandas.tseries.offsets.Hour.is_year_end

Hour.is_year_end()

3.7.31 Minute

Attributes

pandas.tseries.offsets.Minute

class pandas.tseries.offsets.Minute

Attributes

base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Minute.base

Minute.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

__call__(*args, **kwargs)  Call self as a function.

rollback  Roll provided date backward to next offset only if not on offset.

rollforward  Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Minute.__call__

Minute.__call__(*args, **kwargs)
  Call self as a function.

pandas.tseries.offsets.Minute.rollback

Minute.rollback()
  Roll provided date backward to next offset only if not on offset.

  Returns

   timestamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Minute.rollforward

Minute.rollforward()
  Roll provided date forward to next offset only if not on offset.

  Returns

   timestamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.
### Properties

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Minute.delta</code></td>
</tr>
<tr>
<td><code>Minute.freqstr</code></td>
</tr>
<tr>
<td><code>Minute.kwds</code></td>
</tr>
<tr>
<td><code>Minute.name</code></td>
</tr>
<tr>
<td><code>Minute.nanos</code></td>
</tr>
<tr>
<td><code>Minute.normalize</code></td>
</tr>
<tr>
<td><code>Minute.rule_code</code></td>
</tr>
<tr>
<td><code>Minute.n</code></td>
</tr>
</tbody>
</table>

#### pandas.tseries.offsets.Minute.delta

**Minute.delta**

#### pandas.tseries.offsets.Minute.freqstr

**Minute.freqstr**
pandas.tseries.offsets.Minute.kwds

Minute.kwds

pandas.tseries.offsets.Minute.name

Minute.name

pandas.tseries.offsets.Minute.nanos

Minute.nanos

pandas.tseries.offsets.Minute.normalize

Minute.normalize

pandas.tseries.offsets.Minute.rule_code

Minute.rule_code

pandas.tseries.offsets.Minute.n

Minute.n
Methods

Minute.copy
Minute.isAnchored
Minute.onOffset
Minute.is_anchored
Minute.is_on_offset
Minute.__call__(*args, **kwargs) Call self as a function.
Minute.apply
Minute.apply_index(other)
Minute.is_month_start
Minute.is_month_end
Minute.is_quarter_start
Minute.is_quarter_end
Minute.is_year_start
Minute.is_year_end

pandas.tseries.offsets.Minute.copy
Minute.copy()

pandas.tseries.offsets.Minute.isAnchored
Minute.isAnchored()

pandas.tseries.offsets.Minute.onOffset
Minute.onOffset()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Minute.is_anchored

Minute.is_anchored()

pandas.tseries.offsets.Minute.is_on_offset

Minute.is_on_offset()

pandas.tseries.offsets.Minute.apply

Minute.apply()

pandas.tseries.offsets.Minute.apply_index

Minute.apply_index(other)

pandas.tseries.offsets.Minute.is_month_start

Minute.is_month_start()

pandas.tseries.offsets.Minute.is_month_end

Minute.is_month_end()

pandas.tseries.offsets.Minute.is_quarter_start

Minute.is_quarter_start()

pandas.tseries.offsets.Minute.is_quarter_end

Minute.is_quarter_end()

pandas.tseries.offsets.Minute.is_year_start

Minute.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
pandas.tseries.offsets.Minute.is_year_end

Minute.is_year_end()

3.7.32 Second

__Second

Attributes

pandas.tseries.offsets.Second

class pandas.tseries.offsets.Second

Attributes

base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Second.base

Second.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>delta</th>
<th>freqstr</th>
<th>kwds</th>
<th>n</th>
<th>name</th>
<th>nanos</th>
<th>normalize</th>
<th>rule_code</th>
</tr>
</thead>
</table>

Methods

```call__(*args, **kwargs) Call self as a function.

rollback

Roll provided date backward to next offset only if not on offset.

rollforward

Roll provided date forward to next offset only if not on offset.

```

3.7. Date offsets 2471
pandas.tseries.offsets.Second.__call__

Second.__call__(*args, **kwargs)
   Call self as a function.

pandas.tseries.offsets.Second.rollback

Second.rollback()
   Roll provided date backward to next offset only if not on offset.

   Returns

   TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Second.rollforward

Second.rollforward()
   Roll provided date forward to next offset only if not on offset.

   Returns

   TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

+-----------------+-----------------+-----------------+
| apply           | apply_index     | copy            |
| isAnchored      | is_anchored     | is_month_end    |
| is_month_start  | is_on_offset    | is_quarter_end  |
| is_quarter_start| is_year_end     | is_year_start   |
| onOffset        |                 |                 |
+-----------------+-----------------+-----------------+
Properties

- Second.delta
- Second.freqstr
- Second.kwds
- Second.name
- Second.nanos
- Second.normalize
- Second.rule_code
- Second.n

pandas.tseries.offsets.Second.delta

Second.delta

pandas.tseries.offsets.Second.freqstr

Second.freqstr

pandas.tseries.offsets.Second.kwds

Second.kwds

pandas.tseries.offsets.Second.name

Second.name

pandas.tseries.offsets.Second.nanos

Second.nanos
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Second.normalize

Second.normalize

pandas.tseries.offsets.Second.rule_code

Second.rule_code

pandas.tseries.offsets.Second.n

Second.n

Methods

Second.copy

Second.isAnchored

Second.onOffset

Second.is_anchored

Second.is_on_offset

Second.__call__(*args, **kwargs) Call self as a function.

Second.apply

Second.apply_index(other)

Second.is_month_start

Second.is_month_end

Second.is_quarter_start

Second.is_quarter_end

Second.is_year_start

Second.is_year_end
pandas.tseries.offsets.Second.copy
Second.copy()

pandas.tseries.offsets.Second.isAnchored
Second.isAnchored()

pandas.tseries.offsets.Second.onOffset
Second.onOffset()

pandas.tseries.offsets.Second.is_anchored
Second.is_anchored()

pandas.tseries.offsets.Second.is_on_offset
Second.is_on_offset()

pandas.tseries.offsets.Second.apply
Second.apply()

pandas.tseries.offsets.Second.apply_index
Second.apply_index(other)

pandas.tseries.offsets.Second.is_month_start
Second.is_month_start()

pandas.tseries.offsets.Second.is_month_end
Second.is_month_end()
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
pandas.tseries.offsets.Second.is_quarter_start
Second.is_quarter_start()
```

```
pandas.tseries.offsets.Second.is_quarter_end
Second.is_quarter_end()
```

```
pandas.tseries.offsets.Second.is_year_start
Second.is_year_start()
```

```
pandas.tseries.offsets.Second.is_year_end
Second.is_year_end()
```

### 3.7.33 Milli

**Milli**

```
Attributes
```

```
pandas.tseries.offsets.Milli
class pandas.tseries.offsets.Milli
```

**Attributes**

```
base
Returns a copy of the calling offset object with n=1 and all other attributes equal.
```

```
pandas.tseries.offsets.Milli.base
Milli.base
Returns a copy of the calling offset object with n=1 and all other attributes equal.
```
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__call__</code></td>
<td>Call self as a function.</td>
</tr>
<tr>
<td><code>rollback</code></td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td><code>rollforward</code></td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

#### pandas.tseries.offsets.Milli.__call__

`Milli.__call__(args, **kwargs)`

Call self as a function.

#### pandas.tseries.offsets.Milli.rollback

`Milli.rollback()`

Roll provided date backward to next offset only if not on offset.

**Returns**

- **TimeStamp**: Rolled timestamp if not on offset, otherwise unchanged timestamp.

#### pandas.tseries.offsets.Milli.rollforward

`Milli.rollforward()`

Roll provided date forward to next offset only if not on offset.

**Returns**

- **TimeStamp**: Rolled timestamp if not on offset, otherwise unchanged timestamp.
Properties

Milli.delta

Milli.freqstr

Milli.kwds

Milli.name

Milli.nanos

Milli.normalize

Milli.rule_code

Milli.n

pandas.tseries.offsets.Milli.delta

Milli.delta

pandas.tseries.offsets.Milli.freqstr

Milli.freqstr
pandas.tseries.offsets.Milli.kwds

Milli.kwds

pandas.tseries.offsets.Milli.name

Milli.name

pandas.tseries.offsets.Milli.nanos

Milli.nanos

pandas.tseries.offsets.Milli.normalize

Milli.normalize

pandas.tseries.offsets.Milli.rule_code

Milli.rule_code

pandas.tseries.offsets.Milli.n

Milli.n
Methods

Milli.copy

Milli.isAnchored

Milli.onOffset

Milli.is_anchored

Milli.is_on_offset

Milli.__call__(*args, **kwargs)  Call self as a function.

Milli.apply

Milli.apply_index(other)

Milli.is_month_start

Milli.is_month_end

Milli.is_quarter_start

Milli.is_quarter_end

Milli.is_year_start

Milli.is_year_end

pandas.tseries.offsets.Milli.copy

Milli.copy()

pandas.tseries.offsets.Milli.isAnchored

Milli.isAnchored()

pandas.tseries.offsets.Milli.onOffset

Milli.onOffset()
pandas.tseries.offsets.Milli.is_anchored

Milli.is_anchored()

pandas.tseries.offsets.Milli.is_on_offset

Milli.is_on_offset()

pandas.tseries.offsets.Milli.apply

Milli.apply()

pandas.tseries.offsets.Milli.apply_index

Milli.apply_index(*other*)

pandas.tseries.offsets.Milli.is_month_start

Milli.is_month_start()

pandas.tseries.offsets.Milli.is_month_end

Milli.is_month_end()

pandas.tseries.offsets.Milli.is_quarter_start

Milli.is_quarter_start()

pandas.tseries.offsets.Milli.is_quarter_end

Milli.is_quarter_end()

pandas.tseries.offsets.Milli.is_year_start

Milli.is_year_start()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Milli.is_year_end

Milli.is_year_end()

3.7.34 Micro

Micro

Attributes

pandas.tseries.offsets.Micro
class pandas.tseries.offsets.Micro

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Returns a copy of the calling offset object with n=1 and all other attributes equal.</th>
</tr>
</thead>
</table>

pandas.tseries.offsets.Micro.base

Micro.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args,**kwargs)</td>
</tr>
<tr>
<td>rollback</td>
</tr>
<tr>
<td>rollforward</td>
</tr>
</tbody>
</table>
pandas.tseries.offsets.Micro.__call__

Micro.__call__(*args, **kwargs)
    Call self as a function.

pandas.tseries.offsets.Micro.rollback

Micro.rollback()
    Roll provided date backward to next offset only if not on offset.

Returns

    TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

pandas.tseries.offsets.Micro.rollforward

Micro.rollforward()
    Roll provided date forward to next offset only if not on offset.

Returns

    TimeStamp  Rolled timestamp if not on offset, otherwise unchanged timestamp.

<table>
<thead>
<tr>
<th>apply</th>
<th>apply_index</th>
<th>copy</th>
<th>isAnchored</th>
<th>is_anchored</th>
<th>is_month_end</th>
<th>is_month_start</th>
<th>is_on_offset</th>
<th>is_quarter_end</th>
<th>is_quarter_start</th>
<th>is_year_end</th>
<th>is_year_start</th>
<th>onOffset</th>
</tr>
</thead>
</table>

pandas: powerful Python data analysis toolkit, Release 1.4.2

Properties

Micro.delta

Micro.freqstr

Micro.kwds

Micro.name

Micro.nanos

Micro.normalize

Micro.rule_code

Micro.n

pandas.tseries.offsets.Micro.delta

Micro.delta

pandas.tseries.offsets.Micro.freqstr

Micro.freqstr

pandas.tseries.offsets.Micro.kwds

Micro.kwds

pandas.tseries.offsets.Micro.name

Micro.name

pandas.tseries.offsets.Micro.nanos

Micro.nanos
pandas.tseries.offsets.Micro.normalize

Micro.normalize

pandas.tseries.offsets.Micro.rule_code

Micro.rule_code

pandas.tseries.offsets.Micro.n

Micro.n

Methods

- Micro.copy
- Micro.isAnchored
- Micro.onOffset
- Micro.is_anchored
- Micro.is_on_offset
- Micro.__call__(*args, **kwargs) Call self as a function.
- Micro.apply
- Micro.apply_index(other)
- Micro.is_month_start
- Micro.is_month_end
- Micro.is_quarter_start
- Micro.is_quarter_end
- Micro.is_year_start
- Micro.is_year_end
pandas.tseries.offsets.Micro.copy

Micro.copy()

pandas.tseries.offsets.Micro.isAnchored

Micro.isAnchored()

pandas.tseries.offsets.Micro.onOffset

Micro.onOffset()

pandas.tseries.offsets.Micro.is_anchored

Micro.is_anchored()

pandas.tseries.offsets.Micro.is_on_offset

Micro.is_on_offset()

pandas.tseries.offsets.Micro.apply

Micro.apply()

pandas.tseries.offsets.Micro.apply_index

Micro.apply_index(other)

pandas.tseries.offsets.Micro.is_month_start

Micro.is_month_start()

pandas.tseries.offsets.Micro.is_month_end

Micro.is_month_end()
pandas.tseries.offsets.Micro.is_quarter_start

Micro.is_quarter_start()

pandas.tseries.offsets.Micro.is_quarter_end

Micro.is_quarter_end()

pandas.tseries.offsets.Micro.is_year_start

Micro.is_year_start()

pandas.tseries.offsets.Micro.is_year_end

Micro.is_year_end()

3.7.35 Nano

Nano

Attributes

pandas.tseries.offsets.Nano
class pandas.tseries.offsets.Nano

Attributes

base

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Nano.base

Nano.base

Returns a copy of the calling offset object with n=1 and all other attributes equal.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Call self as a function.</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll provided date backward to next offset only if not on offset.</td>
</tr>
<tr>
<td>rollforward</td>
<td>Roll provided date forward to next offset only if not on offset.</td>
</tr>
</tbody>
</table>

**pandas.tseries.offsets.Nano.__call__**

Nano.__call__(*args, **kwargs)
Call self as a function.

**pandas.tseries.offsets.Nano.rollback**

Nano.rollback()
Roll provided date backward to next offset only if not on offset.

*Returns*

**TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.

**pandas.tseries.offsets.Nano.rollforward**

Nano.rollforward()
Roll provided date forward to next offset only if not on offset.

*Returns*

**TimeStamp** Rolled timestamp if not on offset, otherwise unchanged timestamp.
### Properties

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Nano.delta</code></td>
</tr>
<tr>
<td><code>Nano.freqstr</code></td>
</tr>
<tr>
<td><code>Nano.kwds</code></td>
</tr>
<tr>
<td><code>Nano.name</code></td>
</tr>
<tr>
<td><code>Nano.nanos</code></td>
</tr>
<tr>
<td><code>Nano.normalize</code></td>
</tr>
<tr>
<td><code>Nano.rule_code</code></td>
</tr>
<tr>
<td><code>Nano.n</code></td>
</tr>
</tbody>
</table>

#### pandas.tseries.offsets.Nano.delta

`Nano.delta`

#### pandas.tseries.offsets.Nano.freqstr

`Nano.freqstr`
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Nano.kwds
Nano.kwds

pandas.tseries.offsets.Nano.name
Nano.name

pandas.tseries.offsets.Nano.nanos
Nano.nanos

pandas.tseries.offsets.Nano.normalize
Nano.normalize

pandas.tseries.offsets.Nano.rule_code
Nano.rule_code

pandas.tseries.offsets.Nano.n
Nano.n
Methods

Nano.copy
Nano.isAnchored
Nano.onOffset
Nano.is_anchored
Nano.is_on_offset
Nano._call_(*args, **kwargs)  # Call self as a function.
Nano.apply
Nano.apply_index(other)
Nano.is_month_start
Nano.is_month_end
Nano.is_quarter_start
Nano.is_quarter_end
Nano.is_year_start
Nano.is_year_end

pandas.tseries.offsets.Nano.copy
Nanocopy()

pandas.tseries.offsets.Nano.isAnchored
Nano.isAnchored()

pandas.tseries.offsets.Nano.onOffset
NanoonOffset()
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.tseries.offsets.Nano.is_anchored
Nano.is_anchored()

pandas.tseries.offsets.Nano.is_on_offset
Nano.is_on_offset()

pandas.tseries.offsets.Nano.apply
Nano.apply()

pandas.tseries.offsets.Nano.apply_index
Nano.apply_index(other)

pandas.tseries.offsets.Nano.is_month_start
Nano.is_month_start()

pandas.tseries.offsets.Nano.is_month_end
Nano.is_month_end()

pandas.tseries.offsets.Nano.is_quarter_start
Nano.is_quarter_start()

pandas.tseries.offsets.Nano.is_quarter_end
Nano.is_quarter_end()

pandas.tseries.offsets.Nano.is_year_start
Nano.is_year_start()
3.8 Frequencies

| to_offset | Return DateOffset object from string or tuple representation or datetime.timedelta object. |

3.8.1 pandas.tseries.frequencies.to_offset

pandas.tseries.frequencies.to_offset()  
Return DateOffset object from string or tuple representation or datetime.timedelta object.  

Parameters  
freq [str, datetime.timedelta, BaseOffset or None]  

Returns  
DateOffset or None  

Raises  
ValueError If freq is an invalid frequency  

See also:  
BaseOffset Standard kind of date increment used for a date range.

Examples

```python
>>> to_offset("5min")
<5 * Minutes>

>>> to_offset("1D1H")
<25 * Hours>

>>> to_offset("2W")
<2 * Weeks: weekday=6>

>>> to_offset("2B")
<2 * BusinessDays>

>>> to_offset(pd.Timedelta(days=1))
<Day>

>>> to_offset(Hour())
<Hour>
```
3.9 Window

Rolling objects are returned by `.rolling` calls: `pandas.DataFrame.rolling()`, `pandas.Series.rolling()`, etc. Expanding objects are returned by `.expanding` calls: `pandas.DataFrame.expanding()`, `pandas.Series.expanding()`, etc. ExponentialMovingWindow objects are returned by `.ewm` calls: `pandas.DataFrame.ewm()`, `pandas.Series.ewm()`, etc.

3.9.1 Rolling window functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Rolling.count()</code></td>
<td>Calculate the rolling count of non NaN observations.</td>
</tr>
<tr>
<td><code>Rolling.sum(*args[, engine, engine_kwars])</code></td>
<td>Calculate the rolling sum.</td>
</tr>
<tr>
<td><code>Rolling.mean(*args[, engine, engine_kwars])</code></td>
<td>Calculate the rolling mean.</td>
</tr>
<tr>
<td><code>Rolling.median([engine, engine_kwars])</code></td>
<td>Calculate the rolling median.</td>
</tr>
<tr>
<td><code>Rolling.var([ddof, engine, engine_kwars])</code></td>
<td>Calculate the rolling variance.</td>
</tr>
<tr>
<td><code>Rolling.std([ddof, engine, engine_kwars])</code></td>
<td>Calculate the rolling standard deviation.</td>
</tr>
<tr>
<td><code>Rolling.min(*args[, engine, engine_kwars])</code></td>
<td>Calculate the rolling minimum.</td>
</tr>
<tr>
<td><code>Rolling.max(*args[, engine, engine_kwars])</code></td>
<td>Calculate the rolling maximum.</td>
</tr>
<tr>
<td><code>Rolling.corr([other, pairwise, ddof])</code></td>
<td>Calculate the rolling correlation.</td>
</tr>
<tr>
<td><code>Rolling.cov([other, pairwise, ddof])</code></td>
<td>Calculate the rolling sample covariance.</td>
</tr>
<tr>
<td><code>Rolling.skew(**kwargs)</code></td>
<td>Calculate the rolling unbiased skewness.</td>
</tr>
<tr>
<td><code>Rolling.kurt(**kwargs)</code></td>
<td>Calculate the rolling Fisher's definition of kurtosis without bias.</td>
</tr>
<tr>
<td><code>Rolling.apply(func[, raw, engine, ...])</code></td>
<td>Calculate the rolling custom aggregation function.</td>
</tr>
<tr>
<td><code>Rolling.aggregate(func, *args, **kwargs)</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>Rolling.quantile(quantile[, interpolation])</code></td>
<td>Calculate the rolling quantile.</td>
</tr>
<tr>
<td><code>Rolling.sem([ddof])</code></td>
<td>Calculate the rolling standard error of mean.</td>
</tr>
<tr>
<td><code>Rolling.rank([method, ascending, pct])</code></td>
<td>Calculate the rolling rank.</td>
</tr>
</tbody>
</table>

`pandas.core.window.rolling.Rolling.count`

Rolling.count()  
Calculate the rolling count of non NaN observations.

**Returns**

Series or DataFrame  
Return type is the same as the original object with np.float64 dtype.

**See also:**

pandas.Series.rolling  
Calling rolling with Series data.

pandas.DataFrame.rolling  
Calling rolling with DataFrames.

pandas.Series.count  
Aggregating count for Series.

pandas.DataFrame.count  
Aggregating count for DataFrame.
Examples

```python
>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64
```

**pandas.core.window.rolling.Rolling.sum**

Rolling.sum(*args, engine=None, engine_kwars=None, **kwargs*)

Calculate the rolling sum.

**Parameters**

- **args** For NumPy compatibility and will not have an effect on the result.
- **engine** [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.3.0.
- **engine_kwars** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwars
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwars for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}
    New in version 1.3.0.
- **kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

- **Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

**See also:**

- **pandas.Series.rolling** Calling rolling with Series data.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.sum Aggregating sum for Series.
pandas.DataFrame.sum Aggregating sum for DataFrame.
Notes
See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations
for the Numba engine.
Examples
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0
1
1
2
2
3
3
4
4
5
dtype: int64
>>> s.rolling(3).sum()
0
NaN
1
NaN
2
6.0
3
9.0
4
12.0
dtype: float64
>>> s.rolling(3, center=True).sum()
0
NaN
1
6.0
2
9.0
3
12.0
4
NaN
dtype: float64
For DataFrame, each sum is computed column-wise.
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
A
B
0 1
1
1 2
4
2 3
9
3 4 16
4 5 25
>>> df.rolling(3).sum()
A
B
0
NaN
NaN
1
NaN
NaN
(continues on next page)

2496

Chapter 3. API reference


**pandas.core.window.rolling.Rolling.mean**

Rolling.mean(*args, engine=None, engine_kwars=None, **kwargs)

Calculate the rolling mean.

**Parameters**

- *args For NumPy compatibility and will not have an effect on the result.
- engine [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.3.0.
- engine_kwars [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwars
  - For 'numba' engine, the engine can accept nopython, nogil and parallel
dictionary keys. The values must either be True or False. The default
device_kwars for the 'numba' engine is {'nopython': True, 'nogil':
False, 'parallel': False}
    New in version 1.3.0.
- **kwargs For NumPy compatibility and will not have an effect on the result.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- pandas.Series.rolling Calling rolling with Series data.
- pandas.DataFrame.rolling Calling rolling with DataFrames.
- pandas.Series.mean Aggregating mean for Series.
- pandas.DataFrame.mean Aggregating mean for DataFrame.

**Notes**

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.
Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0   NaN
1  1.5
2  2.5
3  3.5
dtype: float64

>>> s.rolling(3).mean()
0   NaN
1  NaN
2  2.0
3  3.0
dtype: float64
```

pandas.core.window.rolling.Rolling.median

`Rolling.median(engine=None, engine_kwarg=None, **kwargs)`

Calculate the rolling median.

**Parameters**

- `engine` [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting `compute.use_numba`

New in version 1.3.0.

- `engine_kwarg` [dict, default None]
  - For 'cython' engine, there are no accepted `engine_kwarg`
  - For 'numba' engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be True or False. The default `engine_kwarg` for the 'numba' engine is {`nopython`: True, `nogil`: False, `parallel`: False}

New in version 1.3.0.

- `**kwargs` For NumPy compatibility and will not have an effect on the result.

**Returns**

- Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- `pandas.Series.rolling` Calling rolling with Series data.
- `pandas.DataFrame.rolling` Calling rolling with DataFrames.
- `pandas.Series.median` Aggregating median for Series.
- `pandas.DataFrame.median` Aggregating median for DataFrame.
Notes

See *Numba engine* and *Numba (JIT compilation)* for extended documentation and performance considerations for the Numba engine.

Examples

Compute the rolling median of a series with a window size of 3.

```python
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
dtype: float64
```

**pandas.core.window.rolling.Rolling.var**

Rolling.var(ddof=1, *args, engine=None, engine_kwargs=None, **kwargs)

Calculate the rolling variance.

**Parameters**

- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.
- **args** For NumPy compatibility and will not have an effect on the result.
- **engine** [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting *compute.use_numba*
    New in version 1.4.0.
- **engine_kwargs** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nogil, parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}
    New in version 1.4.0.
- **kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

*Series or DataFrame* Return type is the same as the original object with *np.float64* dtype.

See also:

- *numpy.var* Equivalent method for NumPy array.
- *pandas.Series.rolling* Calling rolling with Series data.
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.DataFrame.rolling** Calling rolling with DataFrames.
**pandas.Series.var** Aggregating var for Series.
**pandas.DataFrame.var** Aggregating var for DataFrame.

**Notes**

The default `ddof` of 1 used in `Series.var()` is different than the default `ddof` of 0 in `numpy.var()`.

A minimum of one period is required for the rolling calculation.

The implementation is susceptible to floating point imprecision as shown in the example below.

**Examples**

```python
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0          NaN
1          NaN
2  3.333333e-01
3  1.000000e+00
4  1.000000e+00
5  1.333333e+00
6  6.661338e-16
dtype: float64
```

**pandas.core.window.rolling.Rolling.std**

Rolling.std(ddof=1, *args, engine=None, engine_kwargs=None, **kwargs)

Calculate the rolling standard deviation.

**Parameters**

- `ddof` [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - `ddof`, where N represents the number of elements.
- `*args` For NumPy compatibility and will not have an effect on the result.
- `engine` [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting `compute.use_numba`
    New in version 1.4.0.
- `engine_kwargs` [dict, default None]
  - For 'cython' engine, there are no accepted `engine_kwargs`
  - For 'numba' engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be True or False. The default `engine_kwargs` for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}
    New in version 1.4.0.
- `**kwargs` For NumPy compatibility and will not have an effect on the result.
Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

numpy.std  Equivalent method for NumPy array.
pandas.Series.rolling  Calling rolling with Series data.
pandas.DataFrame.rolling  Calling rolling with DataFrames.
pandas.Series.std  Aggregating std for Series.
pandas.DataFrame.std  Aggregating std for DataFrame.

Notes

The default ddof of 1 used in Series.std() is different than the default ddof of 0 in numpy.std().
A minimum of one period is required for the rolling calculation.
The implementation is susceptible to floating point imprecision as shown in the example below.

Examples

```python
code
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0    NaN
1    NaN
2  5.773503e-01
3  1.000000e+00
4  1.000000e+00
5  1.154701e+00
6  2.580957e-08
dtype: float64
```

Pandas.core.window.rolling.Rolling.min

Rolling.min(*args, engine=None, engine_kwags=None, **kwargs)

Calculate the rolling minimum.

Parameters

*args  For NumPy compatibility and will not have an effect on the result.

engine  [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.3.0.

engine_kwags  [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwags
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default

3.9. Window
engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}

New in version 1.3.0.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.rolling Calling rolling with Series data.
pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.min Aggregating min for Series.
pandas.DataFrame.min Aggregating min for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

Examples

Performing a rolling minimum with a window size of 3.

```python
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0    NaN
1    NaN
2    3.0
3    2.0
4    2.0
dtype: float64
```

pandas.core.window.rolling.Rolling.max

Rolling.max(*args, engine=None, engine_kwargs=None, **kwargs)

Calculate the rolling maximum.

Parameters

*args For NumPy compatibility and will not have an effect on the result.

engine [str, default None]

• 'cython' : Runs the operation through C-extensions from cython.
• 'numba' : Runs the operation through JIT compiled code from numba.
• None : Defaults to 'cython' or globally setting compute.use_numba

New in version 1.3.0.

engine_kwargs [dict, default None]

• For 'cython' engine, there are no accepted engine_kwargs
• For 'numba' engine, the engine can accept numpython, nogil and parallel
dictionary keys. The values must either be True or False. The default
engine_kwargs for the 'numba' engine is {'numpython': True, 'nogil':
False, 'parallel': False}

New in version 1.3.0.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64
dtype.

See also:
pandas.Series.rolling Calling rolling with Series data.
pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.max Aggregating max for Series.
pandas.DataFrame.max Aggregating max for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations
for the Numba engine.

pandas.core.window.rolling.Rolling.corr

Rolling.corr(other=None, pairwise=None, ddof=1, **kwargs)

Calculate the rolling correlation.

Parameters

other [Series or DataFrame, optional] If not supplied then will default to self and produce
pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other
will be used and the output will be a DataFrame. If True then all pairwise combina-
tions will be calculated and the output will be a MultiIndexed DataFrame in the case of
DataFrame inputs. In the case of missing elements, only complete pairwise observations
will be used.

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N –
ddof, where N represents the number of elements.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64
dtype.

See also:
cov Similar method to calculate covariance.
numpy.corrcoef NumPy Pearson’s correlation calculation.
pandas.Series.rolling Calling rolling with Series data.
pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.corr Aggregating corr for Series.
pandas.DataFrame.corr Aggregating corr for DataFrame.
Notes

This function uses Pearson’s definition of correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

When other is not specified, the output will be self correlation (e.g. all 1’s), except for DataFrame inputs with pairwise set to True.

Function will return NaN for correlations of equal valued sequences; this is the result of a 0/0 division error.

When pairwise is set to False, only matching columns between self and other will be used.

When pairwise is set to True, the output will be a MultiIndex DataFrame with the original index on the first level, and the other DataFrame columns on the second level.

In the case of missing elements, only complete pairwise observations will be used.

Examples

The below example shows a rolling calculation with a window size of four matching the equivalent function call using numpy.corrcoef().

```python
>>> v1 = [3, 3, 3, 5, 8]
>>> v2 = [3, 4, 4, 4, 8]
>>> # numpy returns a 2X2 array, the correlation coefficient
>>> # is the number at entry [0][1]
>>> print(f"{np.corrcoef(v1[:-1], v2[:-1])[0][1]:.6f}"")
0.333333
>>> print(f"{np.corrcoef(v1[1:], v2[1:])[0][1]:.6f}"")
0.916949
>>> s1 = pd.Series(v1)
>>> s2 = pd.Series(v2)
>>> s1.rolling(4).corr(s2)
0   NaN
1   NaN
2   NaN
3   0.333333
4   0.916949
dtype: float64
```

The below example shows a similar rolling calculation on a DataFrame using the pairwise option.

```python
>>> print(np.corrcoef(matrix[:-1,0], matrix[:-1,1]).round(7))
[1. 0.6263001
 0.6263001 1.]
>>> print(np.corrcoef(matrix[1:,0], matrix[1:,1]).round(7))
[1. 0.5553681
 0.5553681 1.]
>>> df = pd.DataFrame(matrix, columns=['X', 'Y'])
>>> df
   X  Y
0  51.0 35.0
1  49.0 30.0
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

2  47.0  32.0
3  46.0  31.0
4  50.0  36.0

```python
>>> df.rolling(4).corr(pairwise=True)
   X   Y
0 X  NaN  NaN
  Y  NaN  NaN
1 X  NaN  NaN
  Y  NaN  NaN
2 X  NaN  NaN
  Y  NaN  NaN
3 X  1.0000  0.6263
  Y  0.6263  1.0000
4 X  1.0000  0.5554
  Y  0.5554  1.0000
```

### pandas.core.window.rolling.Rolling.cov

Rolling.cov(other=None, pairwise=None, ddof=1, **kwargs)

Calculate the rolling sample covariance.

**Parameters**

- **other** [Series or DataFrame, optional] If not supplied then will default to self and produce pairwise output.
- **pairwise** [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.
- ****kwargs For NumPy compatibility and will not have an effect on the result.

**Returns**

- **Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

See also:

- **pandas.Series.rolling** Calling rolling with Series data.
- **pandas.DataFrame.rolling** Calling rolling with DataFrames.
- **pandas.Series.cov** Aggregating cov for Series.
- **pandas.DataFrame.cov** Aggregating cov for DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.window.rolling.Rolling.skew

Rolling.skew(**kwargs)

Calculate the rolling unbiased skewness.

Parameters

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

scipy.stats.skew Third moment of a probability density.
pandas.Series.rolling Calling rolling with Series data.
pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.skew Aggregating skew for Series.
pandas.DataFrame.skew Aggregating skew for DataFrame.

Notes

A minimum of three periods is required for the rolling calculation.

pandas.core.window.rolling.Rolling.kurt

Rolling.kurt(**kwargs)

Calculate the rolling Fisher's definition of kurtosis without bias.

Parameters

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.rolling Calling rolling with Series data.
pandas.DataFrame.rolling Calling rolling with DataFrames.
pandas.Series.kurt Aggregating kurt for Series.
pandas.DataFrame.kurt Aggregating kurt for DataFrame.

Notes

A minimum of four periods is required for the calculation.
Examples

The example below will show a rolling calculation with a window size of four matching the equivalent function call using scipy.stats.

```python
>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats

>>> print(f"{scipy.stats.kurtosis(arr[:-1], bias=False):.6f}"
-1.200000
>>> print(f"{scipy.stats.kurtosis(arr[1:], bias=False):.6f}"
3.999946
>>> s = pd.Series(arr)
>>> s.rolling(4).kurt()
0       NaN
1       NaN
2       NaN
3   -1.200000
4     3.999946
dtype: float64
```

pandas.core.window.rolling.Rolling.apply

Rolling.apply(func, raw=False, engine=None, engine_kwags=None, args=None, kwags=None)
Calculate the rolling custom aggregation function.

Parameters

- **func** [function] Must produce a single value from an ndarray input if raw=True or a single value from a Series if raw=False. Can also accept a Numba JIT function with engine='numba' specified.
  Changed in version 1.0.0.

- **raw** [bool, default False]
  - False: passes each row or column as a Series to the function.
  - True: the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

- **engine** [str, default None]
  - 'cython': Runs rolling apply through C-extensions from cython.
  - 'numba': Runs rolling apply through JIT compiled code from numba. Only available when raw is set to True.
  - None: Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.0.0.

- **engine_kwags** [dict, default None]
  For 'cython' engine, there are no accepted engine_kwags
  For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwags for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to both the func and the apply rolling aggregation.
New in version 1.0.0.

**args** [tuple, default None] Positional arguments to be passed into func.

**kwargs** [dict, default None] Keyword arguments to be passed into func.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

* pandas.Series.rolling Calling rolling with Series data.
* pandas.DataFrame.rolling Calling rolling with DataFrames.
* pandas.Series.apply Aggregating apply for Series.
* pandas.DataFrame.apply Aggregating apply for DataFrame.

---

**pandas.core.window.rolling.Rolling.aggregate**

Rolling.aggregate(\textit{func}, *\textit{args}, **\textit{kwargs})

Aggregate using one or more operations over the specified axis.

**Parameters**

**func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. \([\text{np.sum}, \ 'mean']\)
- dict of axis labels -> functions, function names or list of such.

**args** Positional arguments to pass to \textit{func}.

**kwargs** Keyword arguments to pass to \textit{func}.

**Returns**

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

* pandas.Series.rolling Calling object with Series data.
* pandas.DataFrame.rolling Calling object with DataFrame data.
Notes

`agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6], "C": [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9

>>> df.rolling(2).sum()
   A  B  C
0  NaN NaN NaN
1  3.0  9.0  15.0
2  5.0  11.0  17.0

>>> df.rolling(2).agg({"A": "sum", "B": "min"})
   A  B
0  NaN NaN
1  3.0  4.0
2  5.0  5.0
```

`pandas.core.window.rolling.Rolling.quantile`  

Rolling.quantile(quantile, interpolation='linear', **kwargs)

Calculate the rolling quantile.

Parameters

- `quantile` [float] Quantile to compute. 0 <= quantile <= 1.
- `interpolation` ["linear", ‘lower’, ‘higher’, ‘midpoint’, ‘nearest’] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points i and j:
  - linear: \( i + (j - i) * fraction \), where fraction is the fractional part of the index surrounded by i and j.
  - lower: i.
  - higher: j.
  - nearest: i or j whichever is nearest.
  - midpoint: \((i + j) / 2\).

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns
Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.rolling  Calling rolling with Series data.
pandas.DataFrame.rolling  Calling rolling with DataFrames.
pandas.Series.quantile  Aggregating quantile for Series.
pandas.DataFrame.quantile  Aggregating quantile for DataFrame.

Examples

```python
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0        NaN
1        1.0
2        2.0
3        3.0
dtype: float64
```

```python
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0       NaN
1      1.5
2      2.5
3      3.5
dtype: float64
```

pandas.core.window.rolling.Rolling.sem

Rolling.sem(ddof=1, *args, **kwargs)

Calculate the rolling standard error of mean.

Parameters

- `ddof` [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \(N - ddof\), where \(N\) represents the number of elements.

  - *args  For NumPy compatibility and will not have an effect on the result.

  - **kwargs  For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.rolling  Calling rolling with Series data.
pandas.DataFrame.rolling  Calling rolling with DataFrames.
pandas.Series.sem  Aggregating sem for Series.
pandas.DataFrame.sem  Aggregating sem for DataFrame.
Notes

A minimum of one period is required for the calculation.

Examples

```python
>>> s = pd.Series([0, 1, 2, 3])
>>> s.rolling(2, min_periods=1).sem()
0      NaN
1    0.707107
2    0.707107
3    0.707107
dtype: float64
```

**pandas.core.window.rolling.Rolling.rank**

`Rolling.rank(method='average', ascending=True, pct=False, **kwargs)`

Calculate the rolling rank.

New in version 1.4.0.

**Parameters**

- **method** [{}'average', 'min', 'max'], default ‘average’] How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group
  - min: lowest rank in the group
  - max: highest rank in the group
- **ascending** [bool, default True] Whether or not the elements should be ranked in ascending order.
- **pct** [bool, default False] Whether or not to display the returned rankings in percentile form.
- **kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

- **Series or DataFrame** Return type is the same as the original object with np. float64 dtype.

See also:

- `pandas.Series.rolling` Calling rolling with Series data.
- `pandas.DataFrame.rolling` Calling rolling with DataFrames.
- `pandas.Series.rank` Aggregating rank for Series.
- `pandas.DataFrame.rank` Aggregating rank for DataFrame.
Examples

```python
>>> s = pd.Series([1, 4, 2, 3, 5, 3])
>>> s.rolling(3).rank()
0    NaN
1    NaN
2    2.0
3    2.0
4    3.0
5    1.5
dtype: float64

>>> s.rolling(3).rank(method="max")
0    NaN
1    NaN
2    2.0
3    2.0
4    3.0
5    2.0
dtype: float64

>>> s.rolling(3).rank(method="min")
0    NaN
1    NaN
2    2.0
3    2.0
4    3.0
5    1.0
dtype: float64
```

3.9.2 Weighted window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window.mean</td>
<td>Calculate the rolling weighted window mean.</td>
</tr>
<tr>
<td>Window.sum</td>
<td>Calculate the rolling weighted window sum.</td>
</tr>
<tr>
<td>Window.var</td>
<td>Calculate the rolling weighted window variance.</td>
</tr>
<tr>
<td>Window.std</td>
<td>Calculate the rolling weighted window standard deviation.</td>
</tr>
</tbody>
</table>

pandas.core.window.rolling.Window.mean

Window.mean(*args, **kwargs)

Calculate the rolling weighted window mean.

Parameters

**kwargs Keyword arguments to configure the SciPy weighted window type.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.rolling Calling rolling with Series data.
**pandas.DataFrame.rolling** Calling rolling with DataFrames.
**pandas.Series.mean** Aggregating mean for Series.
**pandas.DataFrame.mean** Aggregating mean for DataFrame.

**pandas.core.window.rolling.Window.sum**

```
Window.sum(*args, **kwargs)
```

Calculate the rolling weighted window sum.

**Parameters**

**kwargs** Keyword arguments to configure the SciPy weighted window type.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- **pandas.Series.rolling** Calling rolling with Series data.
- **pandas.DataFrame.rolling** Calling rolling with DataFrames.
- **pandas.Series.sum** Aggregating sum for Series.
- **pandas.DataFrame.sum** Aggregating sum for DataFrame.

**pandas.core.window.rolling.Window.var**

```
Window.var(ddof=1, *args, **kwargs)
```

Calculate the rolling weighted window variance.

New in version 1.0.0.

**Parameters**

**kwargs** Keyword arguments to configure the SciPy weighted window type.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- **pandas.Series.rolling** Calling rolling with Series data.
- **pandas.DataFrame.rolling** Calling rolling with DataFrames.
- **pandas.Series.var** Aggregating var for Series.
- **pandas.DataFrame.var** Aggregating var for DataFrame.

**pandas.core.window.rolling.Window.std**

```
Window.std(ddof=1, *args, **kwargs)
```

Calculate the rolling weighted window standard deviation.

New in version 1.0.0.

**Parameters**

**kwargs** Keyword arguments to configure the SciPy weighted window type.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- **pandas.Series.rolling** Calling rolling with Series data.
3.9.3 Expanding window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Expanding.count()</code></td>
<td>Calculate the expanding count of non NaN observations.</td>
</tr>
<tr>
<td><code>Expanding.sum(*args, engine, engine_kwargs)</code></td>
<td>Calculate the expanding sum.</td>
</tr>
<tr>
<td><code>Expanding.mean(*args, engine, engine_kwargs)</code></td>
<td>Calculate the expanding mean.</td>
</tr>
<tr>
<td><code>Expanding.median(engine, engine_kwargs)</code></td>
<td>Calculate the expanding median.</td>
</tr>
<tr>
<td><code>Expanding.var([ddof, engine, engine_kwargs])</code></td>
<td>Calculate the expanding variance.</td>
</tr>
<tr>
<td><code>Expanding.std([ddof, engine, engine_kwargs])</code></td>
<td>Calculate the expanding standard deviation.</td>
</tr>
<tr>
<td><code>Expanding.min(*args, engine, engine_kwargs)</code></td>
<td>Calculate the expanding minimum.</td>
</tr>
<tr>
<td><code>Expanding.max(*args, engine, engine_kwargs)</code></td>
<td>Calculate the expanding maximum.</td>
</tr>
<tr>
<td><code>Expanding.corr([other, pairwise, ddof])</code></td>
<td>Calculate the expanding correlation.</td>
</tr>
<tr>
<td><code>Expanding.cov([other, pairwise, ddof])</code></td>
<td>Calculate the expanding sample covariance.</td>
</tr>
<tr>
<td><code>Expanding.skew(**kwargs)</code></td>
<td>Calculate the expanding unbiased skewness.</td>
</tr>
<tr>
<td><code>Expanding.kurt(**kwargs)</code></td>
<td>Calculate the expanding Fisher's definition of kurtosis without bias.</td>
</tr>
<tr>
<td><code>Expanding.apply(func[, raw, engine, ...])</code></td>
<td>Calculate the expanding custom aggregation function.</td>
</tr>
<tr>
<td><code>Expanding.aggregate(func, *args, **kwargs)</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>Expanding.quantile(quantile[, interpolation])</code></td>
<td>Calculate the expanding quantile.</td>
</tr>
<tr>
<td><code>Expanding.sem([ddof])</code></td>
<td>Calculate the expanding standard error of mean.</td>
</tr>
<tr>
<td><code>Expanding.rank([method, ascending, pct])</code></td>
<td>Calculate the expanding rank.</td>
</tr>
</tbody>
</table>

**pandas.core.window.expanding.Expanding.count**

`Expanding.count()`

Calculate the expanding count of non NaN observations.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

**See also:**

- `pandas.Series.expanding` Calling expanding with Series data.
- `pandas.DataFrame.expanding` Calling expanding with DataFrames.
- `pandas.Series.count` Aggregating count for Series.
- `pandas.DataFrame.count` Aggregating count for DataFrame.

**pandas.core.window.expanding.Expanding.sum**

`Expanding.sum(*args, engine=None, engine_kwargs=None, **kwargs)`

Calculate the expanding sum.

**Parameters**

* `*args` For NumPy compatibility and will not have an effect on the result.

* `engine` [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
pandas: powerful Python data analysis toolkit, Release 1.4.2

- None : Defaults to 'cython' or globally setting compute.use_numba
  New in version 1.3.0.

**kwargs [dict, default None]
- For 'cython' engine, there are no accepted engine_kwargs
- For 'numba' engine, the engine can accept nopython, nogil and parallel
dictionary keys. The values must either be True or False. The default
engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil':
False, 'parallel': False}
  New in version 1.3.0.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64
dtype.

See also:
- pandas.Series.expanding Calling expanding with Series data.
- pandas.DataFrame.expanding Calling expanding with DataFrames.
- pandas.Series.sum Aggregating sum for Series.
- pandas.DataFrame.sum Aggregating sum for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations
for the Numba engine.

pandas.core.window.expanding.Expanding.mean

Expanding.mean(*args, engine=None, engine_kwargs=None, **kwargs)
Calculate the expanding mean.

Parameters

- *args For NumPy compatibility and will not have an effect on the result.
- engine [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.3.0.
- engine_kwargs [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nopython, nogil and parallel
dictionary keys. The values must either be True or False. The default
engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil':
False, 'parallel': False}
    New in version 1.3.0.
- **kwargs For NumPy compatibility and will not have an effect on the result.
Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.expanding Calling expanding with Series data.
pandas.DataFrame.expanding Calling expanding with DataFrames.
pandas.Series.mean  Aggregating mean for Series.
pandas.DataFrame.mean  Aggregating mean for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

pandas.core.window.expanding.Expanding.median

Expanding.median(engine=None, engine_kw_args=None, **kwargs)

Calculate the expanding median.

Parameters

engine  [str, default None]

- 'cython' : Runs the operation through C-extensions from cython.
- 'numba' : Runs the operation through JIT compiled code from numba.
- None : Defaults to 'cython' or globally setting compute.use_numba

  New in version 1.3.0.

engine_kw_args  [dict, default None]

- For 'cython' engine, there are no accepted engine_kw_args
- For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kw_args for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}

  New in version 1.3.0.

**kwargs  For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.expanding Calling expanding with Series data.
pandas.DataFrame.expanding Calling expanding with DataFrames.
pandas.Series.median  Aggregating median for Series.
pandas.DataFrame.median  Aggregating median for DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

Notes

See *Numba engine* and *Numba (JIT compilation)* for extended documentation and performance considerations for the Numba engine.

**pandas.core.window.expanding.Expanding.var**

`Expanding.var(ddof=1, *args, engine=None, engine_kwars=None, **kwargs)`

Calculate the expanding variance.

**Parameters**

- `ddof` [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.
- `*args` For NumPy compatibility and will not have an effect on the result.
- `engine` [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting `compute.use_numba`

  New in version 1.4.0.
- `engine_kwars` [dict, default None]
  - For 'cython' engine, there are no accepted `engine_kwars`
  - For 'numba' engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be True or False. The default `engine_kwars` for the 'numba' engine is `{ 'nopython': True, 'nogil': False, 'parallel': False }`

  New in version 1.4.0.
- `**kwargs` For NumPy compatibility and will not have an effect on the result.

**Returns**

- `Series` or `DataFrame` Return type is the same as the original object with `np.float64` dtype.

See also:

- `numpy.var` Equivalent method for NumPy array.
- `pandas.Series.expanding` Calling expanding with Series data.
- `pandas.DataFrame.expanding` Calling expanding with DataFrames.
- `pandas.Series.var` Aggregating var for Series.
- `pandas.DataFrame.var` Aggregating var for DataFrame.
Notes

The default *ddof* of 1 used in *Series.var()* is different than the default *ddof* of 0 in *numpy.var()*.
A minimum of one period is required for the rolling calculation.

Examples

```python
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])

>>> s.expanding(3).var()
    0   NaN
    1   NaN
    2  0.333333
    3  0.916667
    4  0.800000
    5  0.700000
    6  0.619048
 dtype: float64
```

```
pandas.core.window.expanding.Expanding.std
Expanding.std(ddof=1, *args, engine=None, engine_kwags=None, **kwargs)

Calculate the expanding standard deviation.

Parameters

- *ddof* [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.
- *args* For NumPy compatibility and will not have an effect on the result.
- *engine* [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting compute.use_numba
    New in version 1.4.0.
- *engine_kwags* [dict, default None]
  - For 'cython' engine, there are no accepted *engine_kwags*
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwags for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}
    New in version 1.4.0.
- **kwargs* For NumPy compatibility and will not have an effect on the result.

Returns

- Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:
```
**numpy.std** Equivalent method for NumPy array.

**pandas.Series.expanding** Calling expanding with Series data.

**pandas.DataFrame.expanding** Calling expanding with DataFrames.

**pandas.Series.std** Aggregating std for Series.

**pandas.DataFrame.std** Aggregating std for DataFrame.

**Notes**

The default **ddof** of 1 used in **Series.std()** is different than the default **ddof** of 0 in **numpy.std()**.

A minimum of one period is required for the rolling calculation.

**Examples**

```python
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])

>>> s.expanding(3).std()
0     NaN
1     NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

**pandas.core.window.expanding.Expanding.min**

**Expanding.min**(\*\*args, engine=None, engine_kwargs=None, **kwargs)

Calculate the expanding minimum.

**Parameters**

- \*\*args For NumPy compatibility and will not have an effect on the result.
- **engine** [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting `compute.use_numba`

  New in version 1.3.0.

- **engine_kwargs** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}

  New in version 1.3.0.

- **kwargs** For NumPy compatibility and will not have an effect on the result.
pandas: powerful Python data analysis toolkit, Release 1.4.2

Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.expanding  Calling expanding with Series data.
pandas.DataFrame.expanding  Calling expanding with DataFrames.
pandas.Series.min  Aggregating min for Series.
pandas.DataFrame.min  Aggregating min for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

pandas.core.window.expanding.Expanding.max

Expanding.max(*args, engine=None, engine_kwargs=None, **kwargs)

Calculate the expanding maximum.

Parameters

*args  For NumPy compatibility and will not have an effect on the result.

engine  [str, default None]

• 'cython': Runs the operation through C-extensions from cython.
• 'numba': Runs the operation through JIT compiled code from numba.
• None: Defaults to 'cython' or globally setting compute.use_numba

  New in version 1.3.0.

engine_kwargs  [dict, default None]

  • For 'cython' engine, there are no accepted engine_kwargs
  • For 'numba' engine, the engine can accept nopython, nogil and parallel
dictionary keys. The values must either be True or False. The default
engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}

  New in version 1.3.0.

  **kwargs  For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame  Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.expanding  Calling expanding with Series data.
pandas.DataFrame.expanding  Calling expanding with DataFrames.
pandas.Series.max  Aggregating max for Series.
pandas.DataFrame.max  Aggregating max for DataFrame.
Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

pandas.core.window.expanding.Expanding.corr

Expanding.corr(other=None, pairwise=None, ddof=1, **kwargs)

Calculate the expanding correlation.

Parameters

other [Series or DataFrame, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

cov Similar method to calculate covariance.
numpy.corrcoef NumPy Pearson’s correlation calculation.
pandas.Series.expanding Calling expanding with Series data.
pandas.DataFrame.expanding Calling expanding with DataFrames.
pandas.Series.corr Aggregating corr for Series.
pandas.DataFrame.corr Aggregating corr for DataFrame.

Notes

This function uses Pearson’s definition of correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

When other is not specified, the output will be self correlation (e.g. all 1’s), except for DataFrame inputs with pairwise set to True.

Function will return NaN for correlations of equal valued sequences; this is the result of a 0/0 division error.

When pairwise is set to False, only matching columns between self and other will be used.

When pairwise is set to True, the output will be a MultiIndex DataFrame with the original index on the first level, and the other DataFrame columns on the second level.

In the case of missing elements, only complete pairwise observations will be used.
pandas core window expanding Expanding cov

Expanding cov(\text{other}=\text{None}, \text{pairwise}=\text{None}, \text{ddof}=1, **\text{kwargs})

Calculate the expanding sample covariance.

Parameters

\text{other} \ [\text{Series or DataFrame, optional}] \ If \ not \ supplied \ then \ will \ default \ to \ self \ and \ produce \ pairwise \ output.

\text{pairwise} \ [\text{bool, default None}] \ If \ False \ then \ only \ matching \ columns \ between \ self \ and \ other \ will \ be \ used \ and \ the \ output \ will \ be \ a \ DataFrame. \ If \ True \ then \ all \ pairwise \ combinations \ will \ be \ calculated \ and \ the \ output \ will \ be \ a \ MultiIndexed \ DataFrame \ in \ the \ case \ of \ DataFrame \ inputs. \ In \ the \ case \ of \ missing \ elements, \ only \ complete \ pairwise \ observations \ will \ be \ used.

\text{ddof} \ [\text{int, default 1}] \ Delta \ Degrees \ of \ Freedom. \ The \ divisor \ used \ in \ calculations \ is \ N - \text{ddof}, \ where \ N \ represents \ the \ number \ of \ elements.

**\text{kwargs} \ For \ NumPy \ compatibility \ and \ will \ not \ have \ an \ effect \ on \ the \ result.

Returns

\text{Series or DataFrame} \ Return \ type \ is \ the \ same \ as \ the \ original \ object \ with \ np.float64 \ dtype.

See also:

\text{pandas.Series.expanding} \ Calling \ expanding \ with \ Series \ data.
\text{pandas.DataFrame.expanding} \ Calling \ expanding \ with \ DataFrames.
\text{pandas.Series.cov} \ Aggregating \ cov \ for \ Series.
\text{pandas.DataFrame.cov} \ Aggregating \ cov \ for \ DataFrame.

pandas core window expanding Expanding skew

Expanding skew(**\text{kwargs})

Calculate the expanding unbiased skewness.

Parameters

**\text{kwargs} \ For \ NumPy \ compatibility \ and \ will \ not \ have \ an \ effect \ on \ the \ result.

Returns

\text{Series or DataFrame} \ Return \ type \ is \ the \ same \ as \ the \ original \ object \ with \ np.float64 \ dtype.

See also:

\text{scipy.stats.skew} \ Third \ moment \ of \ a \ probability \ density.
\text{pandas.Series.expanding} \ Calling \ expanding \ with \ Series \ data.
\text{pandas.DataFrame.expanding} \ Calling \ expanding \ with \ DataFrames.
\text{pandas.Series.skew} \ Aggregating \ skew \ for \ Series.
\text{pandas.DataFrame.skew} \ Aggregating \ skew \ for \ DataFrame.
Notes

A minimum of three periods is required for the rolling calculation.

**pandas.core.window.expanding.Expanding.kurt**

```python
Expanding.kurt(**kwargs)
```

Calculate the expanding Fisher’s definition of kurtosis without bias.

**Parameters**

**kwargs For NumPy compatibility and will not have an effect on the result.

**Returns**

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

- pandas.Series.expanding Calling expanding with Series data.
- pandas.DataFrame.expanding Calling expanding with DataFrames.
- pandas.Series.kurt Aggregating kurt for Series.
- pandas.DataFrame.kurt Aggregating kurt for DataFrame.

Notes

A minimum of four periods is required for the calculation.

Examples

The example below will show a rolling calculation with a window size of four matching the equivalent function call using scipy.stats.

```python
>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats
>>> print(f"{scipy.stats.kurtosis(arr[:-1], bias=False):.6f}"")
-1.200000
>>> print(f"{scipy.stats.kurtosis(arr, bias=False):.6f}"")
4.999874
>>> s = pd.Series(arr)
>>> s.expanding(4).kurt()
0      NaN
1      NaN
2      NaN
3    -1.200000
4     4.999874
dtype: float64
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.window.expanding.Expanding.apply

Expanding.apply(func, raw=False, engine=None, engine_kwargs=None, args=None, kwags=None)

Calculate the expanding custom aggregation function.

Parameters

func [function] Must produce a single value from an ndarray input if raw=True or a single value from a Series if raw=False. Can also accept a Numba JIT function with engine='numba' specified.

Changed in version 1.0.0.

raw [bool, default False]

• False : passes each row or column as a Series to the function.
• True : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

engine [str, default None]

• 'cython' : Runs rolling apply through C-extensions from cython.
• 'numba' : Runs rolling apply through JIT compiled code from numba. Only available when raw is set to True.
• None : Defaults to 'cython' or globally setting compute.use_numba

New in version 1.0.0.

engine_kwargs [dict, default None]

• For 'cython' engine, there are no accepted engine_kwargs
• For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to both the func and the apply rolling aggregation.

New in version 1.0.0.

args [tuple, default None] Positional arguments to be passed into func.

kwargs [dict, default None] Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.expanding Calling expanding with Series data.
pandas.DataFrame.expanding Calling expanding with DataFrames.
pandas.Series.apply Aggregating apply for Series.
pandas.DataFrame.apply Aggregating apply for DataFrame.
pandas.core.window.expanding.Expanding.aggregate

Expanding. **aggregate**(func, *args, **kwargs)

Aggregate using one or more operations over the specified axis.

**Parameters**

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.
  
  Accepted combinations are:
  
  - function
  - string function name
  - list of functions and/or function names, e.g. [np.sum, 'mean']
  - dict of axis labels -> functions, function names or list of such.

- **args** Positional arguments to pass to func.

- **kwargs** Keyword arguments to pass to func.

**Returns**

- scalar, Series or DataFrame The return can be:
  
  - scalar : when Series.agg is called with single function
  - Series : when DataFrame.agg is called with a single function
  - DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

**See also:**

- pandas.DataFrame.aggregate Similar DataFrame method.
- pandas.Series.aggregate Similar Series method.

**Notes**

agg is an alias for aggregate. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6], "C": [7, 8, 9]})
>>> df
     A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```
```python
>>> df.ewm(alpha=0.5).mean()
   A    B    C
0  1.000000  4.000000  7.000000
1  1.666667  4.666667  7.666667
2  2.428571  5.428571  8.428571
```

### pandas.core.window.expanding.Expanding.quantile

Expanding.quantile(quantile, interpolation='linear', **kwargs)

Calculate the expanding quantile.

**Parameters**

- **quantile** [float] Quantile to compute. 0 <= quantile <= 1.
- **interpolation** ["linear", "lower", "higher", "midpoint", "nearest"] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points i and j:
  - linear: \( i + (j - i) \times \text{fraction} \), where fraction is the fractional part of the index surrounded by i and j.
  - lower: i.
  - higher: j.
  - nearest: i or j whichever is nearest.
  - midpoint: \( (i + j) / 2 \).

**kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

- **Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

See also:

- [pandas.Series.expanding](#) Calling expanding with Series data.
- [pandas.DataFrame.expanding](#) Calling expanding with DataFrames.
- [pandas.Series.quantile](#) Aggregating quantile for Series.
- [pandas.DataFrame.quantile](#) Aggregating quantile for DataFrame.

### pandas.core.window.expanding.Expanding.sem

Expanding.sem(ddof=1, *args, **kwargs)

Calculate the expanding standard error of mean.

**Parameters**

- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements.

- **args** For NumPy compatibility and will not have an effect on the result.

**kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

- **Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

See also:
**pandas.Series.expanding** Calling expanding with Series data.

**pandas.DataFrame.expanding** Calling expanding with DataFrames.

**pandas.Series.sem** Aggregating sem for Series.

**pandas.DataFrame.sem** Aggregating sem for DataFrame.

### Notes

A minimum of one period is required for the calculation.

### Examples

```python
>>> s = pd.Series([0, 1, 2, 3])

>>> s.expanding().sem()
0     NaN
1  0.707107
2  0.707107
3  0.745356
dtype: float64
```

---

**pandas.core.window.expanding.Expanding.rank**

Expanding.rank*(method='average', ascending=True, pct=False, **kwargs)*

Calculate the expanding rank.

New in version 1.4.0.

#### Parameters

- **method** ([{'average', 'min', 'max'}, default 'average'] How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group
  - min: lowest rank in the group
  - max: highest rank in the group

- **ascending** [bool, default True] Whether or not the elements should be ranked in ascending order.

- **pct** [bool, default False] Whether or not to display the returned rankings in percentile form.

- ****kwargs** For NumPy compatibility and will not have an effect on the result.

#### Returns

- **Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

#### See also:

- **pandas.Series.expanding** Calling expanding with Series data.
- **pandas.DataFrame.expanding** Calling expanding with DataFrames.
- **pandas.Series.rank** Aggregating rank for Series.
- **pandas.DataFrame.rank** Aggregating rank for DataFrame.
Examples

```python
>>> s = pd.Series([1, 4, 2, 3, 5, 3])
>>> s.expanding().rank()
0   1.0
1   2.0
2   2.0
3   3.0
4   5.0
5   3.5
dtype: float64

>>> s.expanding().rank(method="max")
0   1.0
1   2.0
2   2.0
3   3.0
4   5.0
5   4.0
dtype: float64

>>> s.expanding().rank(method="min")
0   1.0
1   2.0
2   2.0
3   3.0
4   5.0
5   3.0
dtype: float64
```

3.9.4 Exponentially-weighted window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ExponentialMovingWindow.mean</code></td>
<td>Calculate the ewm (exponential weighted moment) mean.</td>
</tr>
<tr>
<td><code>ExponentialMovingWindow.sum</code></td>
<td>Calculate the ewm (exponential weighted moment) sum.</td>
</tr>
<tr>
<td><code>ExponentialMovingWindow.std</code></td>
<td>Calculate the ewm (exponential weighted moment) standard deviation.</td>
</tr>
<tr>
<td><code>ExponentialMovingWindow.var</code></td>
<td>Calculate the ewm (exponential weighted moment) variance.</td>
</tr>
<tr>
<td><code>ExponentialMovingWindow.corr</code></td>
<td>Calculate the ewm (exponential weighted moment) sample correlation.</td>
</tr>
<tr>
<td><code>ExponentialMovingWindow.cov</code></td>
<td>Calculate the ewm (exponential weighted moment) sample covariance.</td>
</tr>
</tbody>
</table>
pandas.core.window.ewm.ExponentialMovingWindow.mean

ExponentialMovingWindow.mean(*args, engine=None, engine_kwargs=None, **kwargs)

Calculate the ewm (exponential weighted moment) mean.

Parameters

*args For NumPy compatibility and will not have an effect on the result.

engine [str, default None]

- 'cython': Runs the operation through C-extensions from cython.
- 'numba': Runs the operation through JIT compiled code from numba.
- None: Defaults to 'cython' or globally setting compute.use_numba

New in version 1.3.0.

engine_kwargs [dict, default None]

- For 'cython' engine, there are no accepted engine_kwargs
- For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False}

New in version 1.3.0.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:
pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.mean Aggregating mean for Series.
pandas.DataFrame.mean Aggregating mean for DataFrame.

Notes

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

pandas.core.window.ewm.ExponentialMovingWindow.sum

ExponentialMovingWindow.sum(*args, engine=None, engine_kwargs=None, **kwargs)

Calculate the ewm (exponential weighted moment) sum.

Parameters

*args For NumPy compatibility and will not have an effect on the result.

engine [str, default None]

- 'cython': Runs the operation through C-extensions from cython.
- 'numba': Runs the operation through JIT compiled code from numba.
- None: Defaults to 'cython' or globally setting compute.use_numba
  
  New in version 1.3.0.

**engine_kwargs** [dict, default None]

- For 'cython' engine, there are no accepted engine_kwargs
- For 'numba' engine, the engine can accept nopython, nogil and parallel
dictionary keys. The values must either be True or False. The default
engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil':
False, 'parallel': False}

  New in version 1.3.0.

**kwargs** For NumPy compatibility and will not have an effect on the result.

**Returns**

**Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.sum Aggregating sum for Series.
pandas.DataFrame.sum Aggregating sum for DataFrame.

**Notes**

See Numba engine and Numba (JIT compilation) for extended documentation and performance considerations for the Numba engine.

**pandas.core.window.ewm.ExponentialMovingWindow.std**

ExponentialMovingWindow.std(bias=False, *args, **kwargs)

Calculate the ewm (exponential weighted moment) standard deviation.

**Parameters**

- bias [bool, default False] Use a standard estimation bias correction.

- *args For NumPy compatibility and will not have an effect on the result.

- **kwargs For NumPy compatibility and will not have an effect on the result.

**Returns**

**Series or DataFrame** Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.std Aggregating std for Series.
pandas.DataFrame.std Aggregating std for DataFrame.
pandas.core.window.ewm.ExponentialMovingWindow.var

ExponentialMovingWindow.var(bias=False, *args, **kwargs)

Calculate the ewm (exponential weighted moment) variance.

Parameters

bias [bool, default False] Use a standard estimation bias correction.
*args For NumPy compatibility and will not have an effect on the result.
**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:
pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.var Aggregating var for Series.
pandas.DataFrame.var Aggregating var for DataFrame.

pandas.core.window.ewm.ExponentialMovingWindow.corr

ExponentialMovingWindow.corr(other=None, pairwise=None, **kwargs)

Calculate the ewm (exponential weighted moment) sample correlation.

Parameters

other [Series or DataFrame, optional] If not supplied then will default to self and produce pairwise output.
pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.
**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:
pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.corr Aggregating corr for Series.
pandas.DataFrame.corr Aggregating corr for DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.window.ewm.ExponentialMovingWindow.cov

ExponentialMovingWindow.cov(other=None, pairwise=None, bias=False, **kwargs)

Calculate the ewm (exponential weighted moment) sample covariance.

Parameters

other [Series or DataFrame, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

bias [bool, default False] Use a standard estimation bias correction.

**kwargs For NumPy compatibility and will not have an effect on the result.

Returns

Series or DataFrame Return type is the same as the original object with np.float64 dtype.

See also:

pandas.Series.ewm Calling ewm with Series data.
pandas.DataFrame.ewm Calling ewm with DataFrames.
pandas.Series.cov Aggregating cov for Series.
pandas.DataFrame.cov Aggregating cov for DataFrame.

3.9.5 Window indexer

Base class for defining custom window boundaries.

<table>
<thead>
<tr>
<th>api.indexers.BaseIndexer(index_array, ...)</th>
<th>Base class for window bounds calculations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>api.indexers.FixedForwardWindowIndexer(...)</td>
<td>Creates window boundaries for fixed-length windows that include the current row.</td>
</tr>
<tr>
<td>api.indexers.VariableOffsetWindowIndexer(...)</td>
<td>Calculate window boundaries based on a non-fixed offset such as a BusinessDay.</td>
</tr>
</tbody>
</table>

pandas.api.indexers.BaseIndexer

class pandas.api.indexers.BaseIndexer(index_array=None, window_size=0, **kwargs)

Base class for window bounds calculations.

Methods

get_window_bounds([num_values, min_periods, ...])

Computes the bounds of a window.
pandas.api.indexers.BaseIndexer.get_window_bounds

BaseIndexer.get_window_bounds(num_values=0, min_periods=None, center=None, closed=None)

Computes the bounds of a window.

Parameters

- **num_values** [int, default 0] number of values that will be aggregated over
- **window_size** [int, default 0] the number of rows in a window
- **min_periods** [int, default None] min_periods passed from the top level rolling API
- **center** [bool, default None] center passed from the top level rolling API
- **closed** [str, default None] closed passed from the top level rolling API
- **win_type** [str, default None] win_type passed from the top level rolling API

Returns

A tuple of ndarray[int64]s, indicating the boundaries of each window.

pandas.api.indexers.FixedForwardWindowIndexer

class pandas.api.indexers.FixedForwardWindowIndexer(index_array=None, window_size=0, **kwargs)

Creates window boundaries for fixed-length windows that include the current row.

Examples

```python
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0

>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
   B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```
Methods

get_window_bounds([num_values,  min_periods,  ...])

pandas.api.indexers.FixedForwardWindowIndexer.get_window_bounds

FixedForwardWindowIndexer.get_window_bounds(num_values=0, min_periods=None, center=None, closed=None)

Computes the bounds of a window.

Parameters

num_values  [int, default 0] number of values that will be aggregated over
window_size  [int, default 0] the number of rows in a window
min_periods  [int, default None] min_periods passed from the top level rolling API
center  [bool, default None] center passed from the top level rolling API
closed  [str, default None] closed passed from the top level rolling API

Returns

A tuple of ndarray[int64], indicating the boundaries of each window

pandas.api.indexers.VariableOffsetWindowIndexer

class pandas.api.indexers.VariableOffsetWindowIndexer(index_array=None, window_size=0, index=None, offset=None, **kwargs)

Calculate window boundaries based on a non-fixed offset such as a BusinessDay.

Methods

get_window_bounds([num_values,  min_periods,  ...])

pandas.api.indexers.VariableOffsetWindowIndexer.get_window_bounds

VariableOffsetWindowIndexer.get_window_bounds(num_values=0, min_periods=None, center=None, closed=None)

Computes the bounds of a window.

Parameters

num_values  [int, default 0] number of values that will be aggregated over
window_size  [int, default 0] the number of rows in a window
**min_periods** [int, default None] min_periods passed from the top level rolling API

**center** [bool, default None] center passed from the top level rolling API

**closed** [str, default None] closed passed from the top level rolling API

**win_type** [str, default None] win_type passed from the top level rolling API

**Returns**

A tuple of ndarray[int64], indicating the boundaries of each window

### 3.10 GroupBy

GroupBy objects are returned by groupby calls: `pandas.DataFrame.groupby()`, `pandas.Series.groupby()`, etc.

#### 3.10.1 Indexing, iteration

<table>
<thead>
<tr>
<th>GroupBy.<strong>iter</strong>()</th>
<th>Groupby iterator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupBy.groups</td>
<td>Dict {group name -&gt; group labels}.</td>
</tr>
<tr>
<td>GroupBy.indices</td>
<td>Dict {group name -&gt; group indices}.</td>
</tr>
<tr>
<td>GroupBy.get_group(name[, obj])</td>
<td>Construct DataFrame from group with provided name.</td>
</tr>
</tbody>
</table>

**pandas.core.groupby.GroupBy.__iter__**

GroupBy.__iter__()

Groupby iterator.

**Returns**

Generator yielding sequence of (name, subsetted object) for each group

**pandas.core.groupby.GroupBy.groups**

**property** GroupBy.groups

Dict {group name -> group labels}.

**pandas.core.groupby.GroupBy.indices**

**property** GroupBy.indices

Dict {group name -> group indices}.
pandas.core.groupby.GroupBy.get_group

GroupBy.get_group(name, obj=None)

Construct DataFrame from group with provided name.

Parameters

- **name** [object] The name of the group to get as a DataFrame.
- **obj** [DataFrame, default None] The DataFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used.

Returns

- **group** [same type as obj]

Grouper(*args, **kwargs)

A Grouper allows the user to specify a groupby instruction for an object.

Pandas.Grouper

class pandas.Grouper(*args, **kwargs)

A Grouper allows the user to specify a groupby instruction for an object.

This specification will select a column via the key parameter, or if the level and/or axis parameters are given, a level of the index of the target object.

If axis and/or level are passed as keywords to both Group and groupby, the values passed to Group take precedence.

Parameters

- **key** [str, defaults to None] Groupby key, which selects the grouping column of the target.
- **level** [name/number, defaults to None] The level for the target index.
- **freq** [str / frequency object, defaults to None] This will groupby the specified frequency if the target selection (via key or level) is a datetime-like object. For full specification of available frequencies, please see here.
- **axis** [str, int, defaults to 0] Number/name of the axis.
- **sort** [bool, default to False] Whether to sort the resulting labels.
- **closed** [['left' or 'right']] Closed end of interval. Only when freq parameter is passed.
- **label** [['left' or 'right']] Interval boundary to use for labeling. Only when freq parameter is passed.
- **convention** [['start', 'end', 'e', 's']] If grouper is PeriodIndex and freq parameter is passed.
- **base** [int, default 0] Only when freq parameter is passed. For frequencies that evenly subdivide 1 day, the “origin” of the aggregated intervals. For example, for ‘5min’ frequency, base could range from 0 through 4. Defaults to 0.

Deprecated since version 1.1.0: The new arguments that you should use are ‘offset’ or ‘origin’.

- **loffset** [str, DateOffset, timedelta object] Only when freq parameter is passed.

Deprecated since version 1.1.0: loffset is only working for .resample(...) and not for Group.Grouper(GH28302). However, loffset is also deprecated for .resample(...) See: DataFrame.resample.
origin [Timestamp or str, default ‘start_day’] The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If string, must be one of the following:

- ‘epoch’: origin is 1970-01-01
- ‘start’: origin is the first value of the timeseries
- ‘start_day’: origin is the first day at midnight of the timeseries

New in version 1.1.0.

- ‘end’: origin is the last value of the timeseries
- ‘end_day’: origin is the ceiling midnight of the last day

New in version 1.3.0.

offset [Timedelta or str, default is None] An offset timedelta added to the origin.

New in version 1.1.0.

dropna [bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

New in version 1.2.0.

Returns

A specification for a groupby instruction

Examples

Syntactic sugar for df.groupby('A')

```
>>> df = pd.DataFrame(
...     {
...         "Animal": ["Falcon", "Parrot", "Falcon", "Falcon", "Parrot"],
...         "Speed": [100, 5, 200, 300, 15],
...     }
... )
>>> df
Animal  Speed
0  Falcon  100
1  Parrot   5
2  Falcon  200
3  Falcon  300
4  Parrot   5

>>> df.groupby(pd.Grouper(key="Animal")).mean()
Speed
Animal
Falcon   200.0
Parrot   10.0
```

Specify a resample operation on the column ‘Publish date’

```
>>> df = pd.DataFrame(
...     {
...         "Publish Date": ["2023-01-01", "2023-01-02", "2023-01-03", "2023-01-04", "2023-01-05"],
...         "Speed": [100, 5, 200, 300, 15],
...     }
... )
>>> df
Publish Date  Speed
0  2023-01-01   100
1  2023-01-02    5
2  2023-01-03  200
3  2023-01-04  300
4  2023-01-05   15

>>> df.groupby(pd.Grouper(key="Publish Date", freq="D")).mean()
Speed
Publish Date
2023-01-01   100.0
2023-01-02    5.0
2023-01-03  200.0
2023-01-04  300.0
2023-01-05  15.0
```

(continues on next page)
... "Publish date": [
...     pd.Timestamp("2000-01-02"),
...     pd.Timestamp("2000-01-02"),
...     pd.Timestamp("2000-01-09"),
...     pd.Timestamp("2000-01-16")
... ],
... "ID": [0, 1, 2, 3],
... "Price": [10, 20, 30, 40]
... }

>>> df
<table>
<thead>
<tr>
<th>Publish date</th>
<th>ID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-02</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>2000-01-16</td>
<td>3</td>
<td>40</td>
</tr>
</tbody>
</table>

>>> df.groupby(pd.Grouper(key="Publish date", freq="1W")).mean()
<table>
<thead>
<tr>
<th>Publish date</th>
<th>ID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-02</td>
<td>0.5</td>
<td>15.0</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>2.0</td>
<td>30.0</td>
</tr>
<tr>
<td>2000-01-16</td>
<td>3.0</td>
<td>40.0</td>
</tr>
</tbody>
</table>

If you want to adjust the start of the bins based on a fixed timestamp:

```python
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
```

```python
>>> ts
2000-10-01 23:30:00    0
2000-10-01 23:37:00    3
2000-10-01 23:44:00    6
2000-10-01 23:51:00    9
2000-10-01 23:58:00   12
2000-10-02 00:05:00   15
2000-10-02 00:12:00   18
2000-10-02 00:19:00   21
2000-10-02 00:26:00   24
Freq: 7T, dtype: int64
```

```python
>>> ts.groupby(pd.Grouper(freq='17min')).sum()
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
2000-10-02 00:22:00   24
Freq: 17T, dtype: int64
```

```python
>>> ts.groupby(pd.Grouper(freq='17min', origin='epoch')).sum()
2000-10-01 23:18:00    0
2000-10-01 23:35:00   18
2000-10-01 23:52:00   27
```

(continues on next page)
If you want to adjust the start of the bins with an offset Timedelta, the two following lines are equivalent:

```python
>>> ts.groupby(pd.Grouper(freq='17min', origin='2000-01-01')).sum()
2000-10-01 23:24:00  3
2000-10-01 23:41:00 15
2000-10-01 23:58:00 45
2000-10-02 00:15:00 45
Freq: 17T, dtype: int64
```

```python
>>> ts.groupby(pd.Grouper(freq='17min', offset='23h30min')).sum()
2000-10-01 23:30:00  9
2000-10-01 23:47:00 21
2000-10-02 00:04:00 54
2000-10-02 00:21:00 24
Freq: 17T, dtype: int64
```

To replace the use of the deprecated `base` argument, you can now use `offset`, in this example it is equivalent to have `base=2`:

```python
>>> ts.groupby(pd.Grouper(freq='17min', offset='2min')).sum()
2000-10-01 23:16:00  0
2000-10-01 23:33:00  9
2000-10-01 23:50:00 36
2000-10-02 00:07:00 39
2000-10-02 00:24:00 24
Freq: 17T, dtype: int64
```
Attributes

(ax)
(groups)

3.10.2 Function application

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupBy.apply(func, *args, **kwargs)</td>
<td>Apply function <code>func</code> group-wise and combine the results together.</td>
</tr>
<tr>
<td>GroupBy.agg(func, *args, **kwargs)</td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td>SeriesGroupBy.aggregate([func, engine, ...])</td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td>DataFrameGroupBy.aggregate([func, engine, ...])</td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td>SeriesGroupBy.transform(func, *args[, ...])</td>
<td>Call function producing a like-indexed Series on each group and return a Series having the same indexes as the original object filled with the transformed values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.transform(func, *args[, ...])</td>
<td>Call function producing a like-indexed DataFrame on each group and return a DataFrame having the same indexes as the original object filled with the transformed values.</td>
</tr>
<tr>
<td>GroupBy.pipe(func, *args, **kwargs)</td>
<td>Apply a function <code>func</code> with arguments to this GroupBy object and return the function's result.</td>
</tr>
</tbody>
</table>

pandas.core.groupby.GroupBy.apply

`GroupBy.apply(func, *args, **kwargs)`

Apply function `func` group-wise and combine the results together.

The function passed to `apply` must take a dataframe as its first argument and return a DataFrame, Series or scalar. `apply` will then take care of combining the results back together into a single dataframe or series. `apply` is therefore a highly flexible grouping method.

While `apply` is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods like `agg` or `transform`. Pandas offers a wide range of method that will be much faster than using `apply` for their specific purposes, so try to use them before reaching for `apply`.

**Parameters**

- `func` [callable] A callable that takes a dataframe as its first argument, and returns a dataframe, a series or a scalar. In addition the callable may take positional and keyword arguments.
- `args`, `kwargs` [tuple and dict] Optional positional and keyword arguments to pass to `func`.

**Returns**

- `applied` [Series or DataFrame]

**See also:**

- `pipe` Apply function to the full GroupBy object instead of to each group.
- `aggregate` Apply aggregate function to the GroupBy object.
- `transform` Apply function column-by-column to the GroupBy object.
- `Series.apply` Apply a function to a Series.
**DataFrame.apply** Apply a function to each row or column of a DataFrame.

**Notes**

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed `func`, see the examples below.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

**Examples**

```python
>>> df = pd.DataFrame({'A': 'a a b'.split(),
                    'B': [1, 2, 3],
                    'C': [4, 6, 5]})

>>> g = df.groupby('A')
```

Notice that `g` has two groups, `a` and `b`. Calling `apply` in various ways, we can get different grouping results:

**Example 1:** below the function passed to `apply` takes a DataFrame as its argument and returns a DataFrame. `apply` combines the result for each group together into a new DataFrame:

```python
>>> g[['B', 'C']].apply(lambda x: x / x.sum())
   B    C
0 0.333333 0.4
1 0.666667 0.6
2 1.000000 1.0
```

**Example 2:** The function passed to `apply` takes a DataFrame as its argument and returns a Series. `apply` combines the result for each group together into a new DataFrame. Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed `func`.

```python
>>> g[['B', 'C']].apply(lambda x: x.astype(float).max() - x.min())
   B    C
A  a  1.0  2.0
   b  0.0  0.0
```

**Example 3:** The function passed to `apply` takes a DataFrame as its argument and returns a scalar. `apply` combines the result for each group together into a Series, including setting the index as appropriate:

```python
>>> g.apply(lambda x: x.C.max() - x.B.min())
   A
a  5
b  2
```

dtype: int64
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.GroupBy.agg

GroupBy.agg(func, *args, **kwargs)

pandas.core.groupby.SeriesGroupBy.aggregate

SeriesGroupBy.aggregate(func=None, *args, engine=None, engine_kwargs=None, **kwargs)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:
  • function
  • string function name
  • list of functions and/or function names, e.g. [np.sum, 'mean']
  • dict of axis labels -> functions, function names or list of such.

Can also accept a Numba JIT function with engine='numba' specified. Only passing a single function is supported with this engine.

If the 'numba' engine is chosen, the function must be a user defined function with values and index as the first and second arguments respectively in the function signature. Each group’s index will be passed to the user defined function and optionally available for use.

Changed in version 1.1.0.

*args Positional arguments to pass to func.

engine [str, default None]  
  • 'cython' : Runs the function through C-extensions from cython.  
  • 'numba' : Runs the function through JIT compiled code from numba.  
  • None : Defaults to 'cython' or globally setting compute.use_numba

New in version 1.1.0.

engine_kwargs [dict, default None]  
  • For 'cython' engine, there are no accepted engine_kwargs  
  • For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to the function

New in version 1.1.0.

**kwargs Keyword arguments to be passed into func.

Returns

Series

See also:

Series.groupby.apply Apply function func group-wise and combine the results together.
**Series.groupby.transform** Aggregate using one or more operations over the specified axis.

**Series.aggregate** Transforms the Series on each group based on the given function.

### Notes

When using `engine='numba'`, there will be no “fall back” behavior internally. The group data and group index will be passed as numpy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed `func`, see the examples below.

### Examples

```python
>>> s = pd.Series([1, 2, 3, 4])

>>> s
0    1
1    2
2    3
3    4
dtype: int64

>>> s.groupby([1, 1, 2, 2]).min()
 1    1
2    3
dtype: int64

>>> s.groupby([1, 1, 2, 2]).agg('min')
 1    1
2    3
dtype: int64

>>> s.groupby([1, 1, 2, 2]).agg(['min', 'max'])
   min  max
1    1    2
2    3    4
```

The output column names can be controlled by passing the desired column names and aggregations as keyword arguments.

```python
>>> s.groupby([1, 1, 2, 2]).agg(
...     minimum='min',
...     maximum='max',
... )
   minimum  maximum
1        1        2
2        3        4
```
Changed in version 1.3.0: The resulting dtype will reflect the return value of the aggregating function.

```python
>>> s.groupby([1, 1, 2, 2]).agg(lambda x: x.astype(float).min())
   1  1.0
   2  3.0
dtype: float64
```

**pandas.core.groupby.DataFrameGroupBy.aggregate**

`DataFrameGroupBy.aggregate(func=None, *args, engine=None, engine_kwars=None, **kwargs)`

Aggregate using one or more operations over the specified axis.

**Parameters**

- `func` [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. Accepted combinations are:
  - function
  - string function name
  - list of functions and/or function names, e.g. [np.sum, 'mean']
  - dict of axis labels -> functions, function names or list of such.

Can also accept a Numba JIT function with `engine='numba'` specified. Only passing a single function is supported with this engine.

If the 'numba' engine is chosen, the function must be a user defined function with values and index as the first and second arguments respectively in the function signature. Each group’s index will be passed to the user defined function and optionally available for use.

Changed in version 1.1.0.

- `*args` Positional arguments to pass to func.
- `engine` [str, default None]
  - 'cython': Runs the function through C-extensions from cython.
  - 'numba': Runs the function through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting `compute.use_numba`

New in version 1.1.0.

- `engine_kwars` [dict, default None]
  - For 'cython' engine, there are no accepted `engine_kwars`
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default `engine_kwars` for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to the function

New in version 1.1.0.

- `**kwargs` Keyword arguments to be passed into func.

**Returns**

`DataFrame`
See also:

- `DataFrame.groupby.apply` Apply function `func` group-wise and combine the results together.
- `DataFrame.groupby.transform` Aggregate using one or more operations over the specified axis.
- `DataFrame.aggregate` Transforms the Series on each group based on the given function.

Notes

When using `engine='numba'`, there will be no “fall back” behavior internally. The group data and group index will be passed as numpy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See the documentation on Mutating with User Defined Function (UDF) methods for more details.

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed `func`, see the examples below.

Examples

```python
>>> df = pd.DataFrame(
...     {
...         "A": [1, 1, 2, 2],
...         "B": [1, 2, 3, 4],
...         "C": [0.362838, 0.227877, 1.267767, -0.562860],
...     }
... )

>>> df
   A  B    C
0  1  1  0.362838
1  1  2  0.227877
2  2  3  1.267767
3  2  4 -0.562860

The aggregation is for each column.

```python
>>> df.groupby('A').agg('min')

   B    C
A
   1  0.227877
   2 -0.562860

Multiple aggregations

```python
>>> df.groupby('A').agg(['min', 'max'])

   B    C
  min max  min max
A
   1  2  0.227877  0.362838
   2  4 -0.562860  1.267767

Select a column for aggregation

```
>>> df.groupby('A').B.agg(['min', 'max'])

min  max
A
1  1  2
2  3  4

Different aggregations per column

>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})

B    C
min  max  sum
A
1  1  2  0.590715
2  3  4  0.704907

To control the output names with different aggregations per column, pandas supports “named aggregation”

>>> df.groupby("A").agg({
...     b_min=pd.NamedAgg(column="B", aggfunc="min"),
...     c_sum=pd.NamedAgg(column="C", aggfunc="sum"))

b_min  c_sum
A
1  1  0.590715
2  3  0.704907

- The keywords are the output column names
- The values are tuples whose first element is the column to select and the second element is the aggregation to apply to that column. Pandas provides the pandas.NamedAgg namedtuple with the fields ['column', 'aggfunc'] to make it clearer what the arguments are. As usual, the aggregation can be a callable or a string alias.

See Named aggregation for more.

Changed in version 1.3.0: The resulting dtype will reflect the return value of the aggregating function.

>>> df.groupby("A")[["B"]].agg(lambda x: x.astype(float).min())

B
A
1  1.0
2  3.0

pandas.core.groupby.SeriesGroupBy.transform

SeriesGroupBy.transform(func, *args, engine=None, engine_kwargs=None, **kwargs)

Call function producing a like-indexed Series on each group and return a Series having the same indexes as the original object filled with the transformed values.

Parameters

f [function] Function to apply to each group.

Can also accept a Numba JIT function with engine='numba' specified.

If the 'numba' engine is chosen, the function must be a user defined function with values and index as the first and second arguments respectively in the function sig-
nature. Each group’s index will be passed to the user defined function and optionally available for use.

Changed in version 1.1.0.

*args  Positional arguments to pass to func.

engine  [str, default None]

•  'cython' : Runs the function through C-extensions from cython.
•  'numba' : Runs the function through JIT compiled code from numba.
•  None  : Defaults to 'cython' or the global setting compute.use_numba

New in version 1.1.0.

engine_kwargs  [dict, default None]

•  For 'cython' engine, there are no accepted engine_kwargs
•  For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to the function

New in version 1.1.0.

**kwargs  Keyword arguments to be passed into func.

Returns

Series

See also:

Series.groupby.apply Apply function func group-wise and combine the results together.
Series.groupby.aggregate Aggregate using one or more operations over the specified axis.
Series.transform Call func on self producing a Series with the same axis shape as self.

Notes

Each group is endowed the attribute ‘name’ in case you need to know which group you are working on.

The current implementation imposes three requirements on f:

•  f must return a value that either has the same shape as the input subframe or can be broadcast to the shape of the input subframe. For example, if f returns a scalar it will be broadcast to have the same shape as the input subframe.
•  if this is a DataFrame, f must support application column-by-column in the subframe. If f also supports application to the entire subframe, then a fast path is used starting from the second chunk.
•  f must not mutate groups. Mutation is not supported and may produce unexpected results. See Mutating with User Defined Function (UDF) methods for more details.

When using engine='numba', there will be no “fall back” behavior internally. The group data and group index will be passed as numpy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed func, see the examples below.
Examples

```python
>>> df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', ...
... 'foo', 'bar'],
... 'B': ['one', 'one', 'two', 'three', ...
... 'two', 'two'],
... 'C': [1, 5, 5, 2, 5],
... 'D': [2.0, 5., 8., 1., 2., 9.]})

>>> grouped = df.groupby('A')
>>> grouped.transform(lambda x: (x - x.mean()) / x.std())
   C    D
0 -1.154701 -0.577350
1  0.577350  0.000000
2  0.577350  1.154701
3 -1.154701 -1.000000
4  0.577350 -0.577350
5  0.577350  1.000000

Broadcast result of the transformation

```

```python
>>> grouped.transform(lambda x: x.max() - x.min())
   C    D
0  4  6.0
1  3  8.0
2  4  6.0
3  3  8.0
4  4  6.0
5  3  8.0

```

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed func, for example:

```python
>>> grouped[['C', 'D']].transform(lambda x: x.astype(int).max())
   C    D
0  5  8
1  5  9
2  5  8
3  5  9
4  5  8
5  5  9
```

`pandas.core.groupby.DataFrameGroupBy.transform`

DataGroupBy.transform(func, *args, engine=None, engine_kwargs=None, **kwargs)

Call function producing a like-indexed DataFrame on each group and return a DataFrame having the same indexes as the original object filled with the transformed values.

Parameters

- `f` [function] Function to apply to each group.

  Can also accept a Numba JIT function with engine='numba' specified.

  If the 'numba' engine is chosen, the function must be a user defined function with values and index as the first and second arguments respectively in the function sig-
nature. Each group’s index will be passed to the user defined function and optionally available for use.

Changed in version 1.1.0.

*args Positional arguments to pass to func.

**engine** [str, default None]
- 'cython': Runs the function through C-extensions from cython.
- 'numba': Runs the function through JIT compiled code from numba.
- None: Defaults to 'cython' or the global setting compute.use_numba

New in version 1.1.0.

**engine_kwargs** [dict, default None]
- For 'cython' engine, there are no accepted engine_kwargs
- For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {'nopython': True, 'nogil': False, 'parallel': False} and will be applied to the function

New in version 1.1.0.

**kwargs Keyword arguments to be passed into func.

**Returns**

DataFrame

See also:

**DataFrame.groupby.apply** Apply function func group-wise and combine the results together.

**DataFrame.groupby.aggregate** Aggregate using one or more operations over the specified axis.

**DataFrame.transform** Call func on self producing a DataFrame with the same axis shape as self.

**Notes**

Each group is endowed the attribute ‘name’ in case you need to know which group you are working on.

The current implementation imposes three requirements on f:
- f must return a value that either has the same shape as the input subframe or can be broadcast to the shape of the input subframe. For example, if f returns a scalar it will be broadcast to have the same shape as the input subframe.
- if this is a DataFrame, f must support application column-by-column in the subframe. If f also supports application to the entire subframe, then a fast path is used starting from the second chunk.
- f must not mutate groups. Mutation is not supported and may produce unexpected results. See Mutating with User Defined Function (UDF) methods for more details.

When using **engine=’numba’**, there will be no “fallback” behavior internally. The group data and group index will be passed as numpy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed func, see the examples below.
Examples

```python
df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', ...
... 'two', 'two'],
... 'B': ['one', 'one', 'two', 'three', ...
... 'two', 'two'],
... 'C': [1, 5, 5, 2, 5, 5],
... 'D': [2.0, 5., 8., 1., 2., 9.]})
grouped = df.groupby('A')
grouped.transform(lambda x: (x - x.mean()) / x.std())
```

Broadcast result of the transformation

```python
grouped.transform(lambda x: x.max() - x.min())
```

Changed in version 1.3.0: The resulting dtype will reflect the return value of the passed `func`, for example:

```python
grouped[['C', 'D']].transform(lambda x: x.astype(int).max())
```

pandas.core.groupby.GroupBy.pipe

GroupBy.pipe(func, *args, **kwargs)

Apply a function `func` with arguments to this GroupBy object and return the function’s result.

Use .pipe when you want to improve readability by chaining together functions that expect Series, DataFrames, GroupBy or Resampler objects. Instead of writing

```python
h(g(f(df.groupby('group'))), arg1=a), arg2=b, arg3=c)
```

You can write
which is much more readable.

Parameters

- **func** [callable or tuple of (callable, str)] Function to apply to this GroupBy object or, alternatively, a (callable, data_keyword) tuple where data_keyword is a string indicating the keyword of callable that expects the GroupBy object.
- **args** [iterable, optional] Positional arguments passed into func.
- **kwargs** [dict, optional] A dictionary of keyword arguments passed into func.

Returns

- **object** [the return type of func.]

See also:

- **Series.pipe** Apply a function with arguments to a series.
- **DataFrame.pipe** Apply a function with arguments to a dataframe.
- **apply** Apply function to each group instead of to the full GroupBy object.

Notes

See more here

Examples

```python
>>> df = pd.DataFrame({'A': 'a b a b'.split(), 'B': [1, 2, 3, 4]})
>>> df
  A  B
0 a  1
1 b  2
2 a  3
3 b  4
```

To get the difference between each groups maximum and minimum value in one pass, you can do

```python
>>> df.groupby('A').pipe(lambda x: x.max() - x.min())
  B
A
a  2
b  2
```
3.10.3 Computations / descriptive stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GroupBy.all([skipna])</code></td>
<td>Return True if all values in the group are truthful, else False.</td>
</tr>
<tr>
<td><code>GroupBy.any([skipna])</code></td>
<td>Return True if any value in the group is truthful, else False.</td>
</tr>
<tr>
<td><code>GroupBy.bfill([limit])</code></td>
<td>Backward fill the values.</td>
</tr>
<tr>
<td><code>GroupBy.backfill([limit])</code></td>
<td>Backward fill the values.</td>
</tr>
<tr>
<td><code>GroupBy.count()</code></td>
<td>Compute count of group, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.cumcount([ascending])</code></td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
<tr>
<td><code>GroupBy.cummax([axis])</code></td>
<td>Cumulative max for each group.</td>
</tr>
<tr>
<td><code>GroupBy.cummin([axis])</code></td>
<td>Cumulative min for each group.</td>
</tr>
<tr>
<td><code>GroupBy.cumprod([axis])</code></td>
<td>Cumulative product for each group.</td>
</tr>
<tr>
<td><code>GroupBy.cumsum([axis])</code></td>
<td>Cumulative sum for each group.</td>
</tr>
<tr>
<td><code>GroupBy.ffill([limit])</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>GroupBy.first([numeric_only, min_count])</code></td>
<td>Compute first of group values.</td>
</tr>
<tr>
<td><code>GroupBy.head([n])</code></td>
<td>Return first n rows of each group.</td>
</tr>
<tr>
<td><code>GroupBy.last([numeric_only, min_count])</code></td>
<td>Compute last of group values.</td>
</tr>
<tr>
<td><code>GroupBy.max([numeric_only, min_count])</code></td>
<td>Compute max of group values.</td>
</tr>
<tr>
<td><code>GroupBy.mean([numeric_only, engine, ...])</code></td>
<td>Compute mean of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.median([numeric_only])</code></td>
<td>Compute median of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.min([numeric_only, engine, ...])</code></td>
<td>Compute min of group values.</td>
</tr>
<tr>
<td><code>GroupBy.ngroup([ascending])</code></td>
<td>Number each group from 0 to the number of groups - 1.</td>
</tr>
<tr>
<td><code>GroupBy.nth(n[, dropna])</code></td>
<td>Take the nth row from each group if n is an int, otherwise a subset of rows.</td>
</tr>
<tr>
<td><code>GroupBy.ohlc()</code></td>
<td>Compute open, high, low and close values of a group, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.pad([limit])</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>GroupBy.prod([numeric_only, min_count])</code></td>
<td>Compute prod of group values.</td>
</tr>
<tr>
<td><code>GroupBy.rank([method, ascending, na_option, ...])</code></td>
<td>Provide the rank of values within each group.</td>
</tr>
<tr>
<td><code>GroupBy.pct_change([periods, fill_method, ...])</code></td>
<td>Calculate pct_change of each value to previous entry in group.</td>
</tr>
<tr>
<td><code>GroupBy.size()</code></td>
<td>Compute group sizes.</td>
</tr>
<tr>
<td><code>GroupBy.sem([ddof])</code></td>
<td>Compute standard error of the mean of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.std([ddof, engine, engine_kwargs])</code></td>
<td>Compute standard deviation of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.sum([numeric_only, min_count, ...])</code></td>
<td>Compute sum of group values.</td>
</tr>
<tr>
<td><code>GroupBy.var([ddof, engine, engine_kwargs])</code></td>
<td>Compute variance of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>GroupBy.tail([n])</code></td>
<td>Return last n rows of each group.</td>
</tr>
</tbody>
</table>
pandas.core.groupby.GroupBy.all

```
final GroupBy.all(skipna=True)
```

Return True if all values in the group are truthful, else False.

**Parameters**

- `skipna` [bool, default True] Flag to ignore nan values during truth testing.

**Returns**

- **Series or DataFrame** DataFrame or Series of boolean values, where a value is True if all elements are True within its respective group, False otherwise.

**See also:**
- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.any

```
final GroupBy.any(skipna=True)
```

Return True if any value in the group is truthful, else False.

**Parameters**

- `skipna` [bool, default True] Flag to ignore nan values during truth testing.

**Returns**

- **Series or DataFrame** DataFrame or Series of boolean values, where a value is True if any element is True within its respective group, False otherwise.

**See also:**
- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.bfill

```
final GroupBy.bfill(limit=None)
```

Backward fill the values.

**Parameters**

- `limit` [int, optional] Limit of how many values to fill.

**Returns**

- **Series or DataFrame** Object with missing values filled.

**See also:**
- `Series.bfill` Backward fill the missing values in the dataset.
- `DataFrame.bfill` Backward fill the missing values in the dataset.
- `Series.fillna` Fill NaN values of a Series.
- `DataFrame.fillna` Fill NaN values of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.GroupBy.backfill

GroupBy.backfill(limit=None)
Backward fill the values.

Parameters

limit [int, optional] Limit of how many values to fill.

Returns

Series or DataFrame Object with missing values filled.

See also:
Series.bfill Backward fill the missing values in the dataset.
DataFrame.bfill Backward fill the missing values in the dataset.
Series.fillna Fill NaN values of a Series.
DataFrame.fillna Fill NaN values of a DataFrame.

pandas.core.groupby.GroupBy.count

final GroupBy.count()
Compute count of group, excluding missing values.

Returns

Series or DataFrame Count of values within each group.

See also:
Series.groupby Apply a function groupby to a Series.
DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.cumcount

final GroupBy.cumcount(ascending=True)
Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))
```

Parameters

ascending [bool, default True] If False, number in reverse, from length of group - 1 to 0.

Returns

Series Sequence number of each element within each group.

See also:
ngroup Number the groups themselves.
Examples

```python
>>> df = pd.DataFrame([[a], [a], [a], [b], [b], [a]],
                    columns=['A'])
>>> df
    A
0  a
1  a
2  b
3  b
4  a
5  a
```

```python
>>> df.groupby('A').cumcount()
0  0
1  1
2  2
3  0
4  1
5  3
dtype: int64
```

```python
>>> df.groupby('A').cumcount(ascending=False)
0  3
1  2
2  1
3  1
4  0
5  0
dtype: int64
```

**pandas.core.groupby.GroupBy.cummax**

`final GroupBy.cummax(axis=0, **kwargs)`

Cumulative max for each group.

**Returns**

Series or DataFrame

See also:

- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

**pandas.core.groupby.GroupBy.cummin**

`final GroupBy.cummin(axis=0, **kwargs)`

Cumulative min for each group.

**Returns**

Series or DataFrame

See also:

- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.
pandas.core.groupby.GroupBy.cumprod

**final** GroupBy.cumprod(*axis=0, *args, **kwargs*)
Cumulative product for each group.

**Returns**
Series or DataFrame

**See also:**
- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.cumsum

**final** GroupBy.cumsum(*axis=0, *args, **kwargs*)
Cumulative sum for each group.

**Returns**
Series or DataFrame

**See also:**
- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.ffill

**final** GroupBy.ffill(*limit=None*)
Forward fill the values.

**Parameters**
- limit [int, optional] Limit of how many values to fill.

**Returns**
Series or DataFrame Object with missing values filled.

**See also:**
- Series.ffill Returns Series with minimum number of char in object.
- DataFrame.ffill Object with missing values filled or None if inplace=True.
- Series.fillna Fill NaN values of a Series.
- DataFrame.fillna Fill NaN values of a DataFrame.

pandas.core.groupby.GroupBy.first

**final** GroupBy.first(*numeric_only=False, min_count=-1*)
Compute first of group values.

**Parameters**
- numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.
- min_count [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**Returns**
Series or DataFrame Computed first of values within each group.
**pandas.core.groupby.GroupBy.head**

**final** GroupBy . head ( n=5 )

Return first n rows of each group.

Similar to . apply ( lambda  x :  x . head ( n ) ), but it returns a subset of rows from the original DataFrame with original index and order preserved ( as_index flag is ignored ) .

**Parameters**

- **n** [int] If positive: number of entries to include from start of each group. If negative: number of entries to exclude from end of each group.

**Returns**

Series or DataFrame  Subset of original Series or DataFrame as determined by n.

See also:

Series . groupby  Apply a function groupby to a Series.

DataFrame . groupby  Apply a function groupby to each row or column of a DataFrame.

**Examples**

```python
>>> df = pd.DataFrame([[1, 2], [1, 4], [5, 6]],
   ... columns=['A', 'B'])
>>> df.groupby('A').head(1)
A  B
0  1  2
2  5  6
>>> df.groupby('A').head(-1)
A  B
0  1  2
```

**pandas.core.groupby.GroupBy.last**

**final** GroupBy . last ( numeric_only=False, min_count=-1 )

Compute last of group values.

**Parameters**

- **numeric_only** [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

- **min_count** [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**Returns**

Series or DataFrame  Computed last of values within each group.
pandas.core.groupby.GroupBy.max

**Final GroupBy.max**(numeric_only=False, min_count=-1)

Compute max of group values.

- **numeric_only** [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.
- **min_count** [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**Returns**

Series or DataFrame Computed max of values within each group.

pandas.core.groupby.GroupBy.mean

**Final GroupBy.mean**(numeric_only=None, engine=None, engine_kwargs=None)

Compute mean of groups, excluding missing values.

- **numeric_only** [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.
- **engine** [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting compute.use_numba

New in version 1.4.0.

- **engine_kwargs** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {{'nopython': True, 'nogil': False, 'parallel': False}}

New in version 1.4.0.

**Returns**

pandas.Series or pandas.DataFrame

See also:

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.
Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
                      'B': [np.nan, 2, 3, 4, 5],
                      'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```python
>>> df.groupby('A').mean()
   B     C
A  
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```python
>>> df.groupby(['A', 'B']).mean()
   C
A B
  
1 2.0 2.0
  4.0 1.0
2 3.0 1.0
  5.0 2.0
```

Groupby one column and return the mean of only particular column in the group.

```python
>>> df.groupby('A')['B'].mean()
A
  
1 3.0
2 4.0
Name: B, dtype: float64
```

```
pandas.core.groupby.GroupBy.median
```

**final GroupBy.median**

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

**Parameters**

- `numeric_only [bool, default True]` Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

**Returns**

- `Series or DataFrame` Median of values within each group.

**See also:**

- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.GroupBy.min

**final** GroupBy.min( numeric_only=False, min_count=- 1 )
Compute min of group values.

**Parameters**

numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**Returns**

Series or DataFrame Computed min of values within each group.

pandas.core.groupby.GroupBy.ngroup

**final** GroupBy.ngroup( ascending=True )
Number each group from 0 to the number of groups - 1.

This is the enumerative complement of cumcount. Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

**Parameters**

ascending [bool, default True] If False, number in reverse, from number of group - 1 to 0.

**Returns**

Series Unique numbers for each group.

**See also:**

cumcount Number the rows in each group.

**Examples**

```python
>>> df = pd.DataFrame({'A': list('aabba')})
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').ngroup()
0 0
1 0
2 0
3 1
4 1
5 0
dtype: int64
>>> df.groupby('A').ngroup(ascending=False)
0 1
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

1 1
2 1
3 0
4 0
5 1
dtype: int64

>>> df.groupby(["A", [1,1,2,3,2,1]]).ngroup()
0 0
1 0
2 1
3 3
4 2
5 0
dtype: int64

pandas.core.groupby.GroupBy.nth

final GroupBy.nth(n, dropna=None)
Take the nth row from each group if n is an int, otherwise a subset of rows.

Can be either a call or an index. dropna is not available with index notation. Index notation accepts a comma separated list of integers and slices.

If dropna, will take the nth non-null row, dropna is either ‘all’ or ‘any’; this is equivalent to calling dropna(how=dropna) before the groupby.

Parameters

- n [int, slice or list of ints and slices] A single nth value for the row or a list of nth values or slices.

  Changed in version 1.4.0: Added slice and lists containing slices. Added index notation.

- dropna [{‘any’, ‘all’, None}, default None] Apply the specified dropna operation before counting which row is the nth row. Only supported if n is an int.

Returns

- Series or DataFrame N-th value within each group.

See also:

Series.groupby Apply a function groupby to a Series.
DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 1],
...                      'B': [np.nan, 2, 3, 4, 5]}, columns=['A', 'B'])
>>> g = df.groupby('A')
>>> g.nth(0)
    B
   A
1 NaN
2 3.0
(continues on next page)
```
Index notation may also be used

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Specifying `dropna` allows count ignoring NaN

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
</tr>
</tbody>
</table>

NaNs denote group exhausted when using `dropna`

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Specifying `as_index=False` in `groupby` keeps the original index.

```python
>>> df.groupby('A', as_index=False).nth(1)
                    A   B
1          1  2.0
4          2  5.0
```

### pandas.core.groupby.GroupBy.ohlc

#### `final` GroupBy.ohlc()

Compute open, high, low and close values of a group, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

**Returns**

- **DataFrame** Open, high, low and close values within each group.

**See also:**

- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.

### pandas.core.groupby.GroupBy.pad

#### `GroupBy.pad(limit=None)`

Forward fill the values.

**Parameters**

- **limit** [int, optional] Limit of how many values to fill.

**Returns**

- **Series or DataFrame** Object with missing values filled.

**See also:**

- `Series.ffill` Returns Series with minimum number of char in object.
- `DataFrame.ffill` Object with missing values filled or None if inplace=True.
- `Series.fillna` Fill NaN values of a Series.
- `DataFrame.fillna` Fill NaN values of a DataFrame.

### pandas.core.groupby.GroupBy.prod

#### `final` GroupBy.prod(*numeric_only=NoDefault.no_default, min_count=0*)

Compute prod of group values.

**Parameters**

- **numeric_only** [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

- **min_count** [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

**Returns**
Series or DataFrame  Computed prod of values within each group.

pandas.core.groupby.GroupBy.rank

**final** GroupBy.rank*(method='average', ascending=True, na_option='keep', pct=False, axis=0)*

Provide the rank of values within each group.

**Parameters**

- **method** ([{'average', 'min', 'max', 'first', 'dense'}, default 'average']
  - average: average rank of group.
  - min: lowest rank in group.
  - max: highest rank in group.
  - first: ranks assigned in order they appear in the array.
  - dense: like ‘min’, but rank always increases by 1 between groups.

- **ascending** [bool, default True] False for ranks by high (1) to low (N).

- **na_option** [{'keep', 'top', 'bottom'}, default 'keep']
  - keep: leave NA values where they are.
  - top: smallest rank if ascending.
  - bottom: smallest rank if descending.

- **pct** [bool, default False] Compute percentage rank of data within each group.

- **axis** [int, default 0] The axis of the object over which to compute the rank.

**Returns**

DataFrame with ranking of values within each group

**See also:**

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.

**Examples**

```python
def = pd.DataFrame(
  ... {
  ...   "group": ["a", "a", "a", "a", "b", "b", "b", "b"],
  ...   "value": [2, 4, 2, 3, 5, 1, 4, 1, 5],
  ... }
  ... )
def
  group  value
  0   a  2
  1   a  4
  2   a  2
  3   a  3
  4   a  5
  5   b  1
  6   b  2
```

(continues on next page)
>>> for method in ['average', 'min', 'max', 'dense', 'first']:
...     df[f'{method}_rank'] = df.groupby('group')['value'].rank(method)

<table>
<thead>
<tr>
<th></th>
<th>group</th>
<th>value</th>
<th>average_rank</th>
<th>min_rank</th>
<th>max_rank</th>
<th>dense_rank</th>
<th>first_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>2</td>
<td>1.5</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>4</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>2</td>
<td>1.5</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>3</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>5</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>5.0</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>1</td>
<td>1.5</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>2</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>4</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>1</td>
<td>1.5</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>5</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

**pandas.core.groupby.GroupBy.pct_change**

**final** GroupBy.pct_change(periods=1, fill_method='ffill', limit=None, freq=None, axis=0)

Calculate pct_change of each value to previous entry in group.

*Returns*

**Series or DataFrame** Percentage changes within each group.

*See also:*

**Series.groupby** Apply a function groupby to a Series.

**DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.

**pandas.core.groupby.GroupBy.size**

**final** GroupBy.size()

Compute group sizes.

*Returns*

**DataFrame or Series** Number of rows in each group as a Series if as_index is True or a

**See also:**

**Series.groupby** Apply a function groupby to a Series.

**DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.
**pandas.core.groupby.GroupBy.sem**

**final** GroupBy.sem(ddof=1)

Compute standard error of the mean of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

**Parameters**

- **ddof** [int, default 1] Degrees of freedom.

**Returns**

- **Series or DataFrame** Standard error of the mean of values within each group.

**See also:**

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.

**pandas.core.groupby.GroupBy.std**

**final** GroupBy.std(ddof=1, engine=None, engine_kwargs=None)

Compute standard deviation of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

**Parameters**

- **ddof** [int, default 1] Degrees of freedom.
- **engine** [str, default None]
  - 'cython' : Runs the operation through C-extensions from cython.
  - 'numba' : Runs the operation through JIT compiled code from numba.
  - None : Defaults to 'cython' or globally setting compute.use_numba

New in version 1.4.0.

- **engine_kwargs** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {{'nopython': True, 'nogil': False, 'parallel': False}}

New in version 1.4.0.

**Returns**

- **Series or DataFrame** Standard deviation of values within each group.

**See also:**

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

**pandas.core.groupby.GroupBy.sum**

```python
final GroupBy.sum(numeric_only=NoDefault.no_default, min_count=0, engine=None, engine_kwargs=None)
```
Compute sum of group values.

**Parameters**
- `numeric_only` [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.
- `min_count` [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

**Returns**
- **Series or DataFrame** Computed sum of values within each group.

**pandas.core.groupby.GroupBy.var**

```python
final GroupBy.var(ddof=1, engine=None, engine_kwargs=None)
```
Compute variance of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

**Parameters**
- `ddof` [int, default 1] Degrees of freedom.
- `engine` [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting `compute.use_numba`

New in version 1.4.0.

- `engine_kwargs` [dict, default None]
  - For 'cython' engine, there are no accepted `engine_kwargs`
  - For 'numba' engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be True or False. The default `engine_kwargs` for the 'numba' engine is `{'nopython': True, 'nogil': False, 'parallel': False}`

New in version 1.4.0.

**Returns**
- **Series or DataFrame** Variance of values within each group.

**See also:**
- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.GroupBy.tail

**final GroupBy.tail(n=5)**

Return last n rows of each group.

Similar to .apply(lambda x: x.tail(n)), but it returns a subset of rows from the original DataFrame with original index and order preserved (as_index flag is ignored).

**Parameters**

- **n** [int] If positive: number of entries to include from end of each group. If negative: number of entries to exclude from start of each group.

**Returns**

- **Series or DataFrame** Subset of original Series or DataFrame as determined by n.

See also:

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.

**Examples**

```python
def = pd.DataFrame([[a, 1], [a, 2], [b, 1], [b, 2]], columns=['A', 'B'])
def.groupby('A').tail(1)
A  B
1  a  2
3  b  2
def.groupby('A').tail(-1)
A  B
1  a  2
3  b  2
```

The following methods are available in both SeriesGroupBy and DataFrameGroupBy objects, but may differ slightly, usually in that the DataFrameGroupBy version usually permits the specification of an axis argument, and often an argument indicating whether to restrict application to columns of a specific data type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrameGroupBy.all([skipna])</td>
<td>Return True if all values in the group are truthful, else False.</td>
</tr>
<tr>
<td>DataFrameGroupBy.any([skipna])</td>
<td>Return True if any value in the group is truthful, else False.</td>
</tr>
<tr>
<td>DataFrameGroupBy.backfill([limit])</td>
<td>Backward fill the values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.bfill([limit])</td>
<td>Backward fill the values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.corr</td>
<td>Compute pairwise correlation of columns, excluding NA/null values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.count()</td>
<td>Compute count of group, excluding missing values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.cov</td>
<td>Compute pairwise covariance of columns, excluding NA/null values.</td>
</tr>
<tr>
<td>DataFrameGroupBy.cumcount([ascending])</td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
<tr>
<td>DataFrameGroupBy.cummax([axis])</td>
<td>Cumulative max for each group.</td>
</tr>
<tr>
<td>DataFrameGroupBy.cummin([axis])</td>
<td>Cumulative min for each group.</td>
</tr>
<tr>
<td>DataFrameGroupBy.cumprod([axis])</td>
<td>Cumulative product for each group.</td>
</tr>
</tbody>
</table>

continues on next page
**Table 12 – continued from previous page**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrameGroupBy.cumsum([axis])</code></td>
<td>Cumulative sum for each group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.describe(**kwargs)</code></td>
<td>Generate descriptive statistics.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.diff()</code></td>
<td>First discrete difference of element.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.ffill([limit])</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.fillna()</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.filter(func[, dropna])</code></td>
<td>Return a copy of a DataFrame excluding filtered elements.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.hist()</code></td>
<td>Make a histogram of the DataFrame's columns.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.idxmax([axis, skipna])</code></td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.idxmin([axis, skipna])</code></td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.mad()</code></td>
<td>Return the mean absolute deviation of the values over the requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.nunique([dropna])</code></td>
<td>Return DataFrame with counts of unique elements in each position.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.pad([limit])</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.pct_change([periods, ...])</code></td>
<td>Calculate pct_change of each value to previous entry in group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.plot()</code></td>
<td>Class implementing the .plot attribute for groupby objects.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.quantile([q, interpolation])</code></td>
<td>Return group values at the given quantile, a la numpy.percentile.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.rank([method, ascending, ...])</code></td>
<td>Provide the rank of values within each group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.resample(rule, *args, **kwargs)</code></td>
<td>Provide resampling when using a TimeGrouper.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.sample([n, frac, replace, ...])</code></td>
<td>Return a random sample of items from each group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.size()</code></td>
<td>Compute group sizes.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.skew()</code></td>
<td>Return unbiased skew over requested axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.take()</code></td>
<td>Return the elements in the given positional indices along an axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.tshift</code></td>
<td>(DEPRECATED) Shift the time index, using the index's frequency if available.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.value_counts([subset, ...])</code></td>
<td>Return a Series or DataFrame containing counts of unique rows.</td>
</tr>
</tbody>
</table>

---

**pandas.core.groupby.DataFrameGroupBy.all**

`DataFrameGroupBy.all(skipna=True)`

Return True if all values in the group are truthful, else False.

**Parameters**

- `skipna` [bool, default True] Flag to ignore nan values during truth testing.

**Returns**

Series or DataFrame  DataFrame or Series of boolean values, where a value is True if all elements are True within its respective group, False otherwise.

**See also:**

- Series.groupby: Apply a function groupby to a Series.
- DataFrame.groupby: Apply a function groupby to each row or column of a DataFrame.
pandas.core.groupby.DataFrameGroupBy.any

DataFrameGroupBy.\texttt{any}(skipna=True)
Return True if any value in the group is truthful, else False.

\textbf{Parameters}

\texttt{skipna} [bool, default True] Flag to ignore nan values during truth testing.

\textbf{Returns}

\textbf{Series or DataFrame} DataFrame or Series of boolean values, where a value is True if any element is True within its respective group, False otherwise.

\textbf{See also:}

\texttt{Series.groupby} Apply a function groupby to a Series.
\texttt{DataFrame.groupby} Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.DataFrameGroupBy.backfill

DataFrameGroupBy.\texttt{backfill}(limit=None)
Backward fill the values.

\textbf{Parameters}

\texttt{limit} [int, optional] Limit of how many values to fill.

\textbf{Returns}

\textbf{Series or DataFrame} Object with missing values filled.

\textbf{See also:}

\texttt{Series.bfill} Backward fill the missing values in the dataset.
\texttt{DataFrame.bfill} Backward fill the missing values in the dataset.
\texttt{Series.fillna} Fill NaN values of a Series.
\texttt{DataFrame.fillna} Fill NaN values of a DataFrame.

pandas.core.groupby.DataFrameGroupBy.bfill

DataFrameGroupBy.\texttt{bfill}(limit=None)
Backward fill the values.

\textbf{Parameters}

\texttt{limit} [int, optional] Limit of how many values to fill.

\textbf{Returns}

\textbf{Series or DataFrame} Object with missing values filled.

\textbf{See also:}

\texttt{Series.bfill} Backward fill the missing values in the dataset.
\texttt{DataFrame.bfill} Backward fill the missing values in the dataset.
\texttt{Series.fillna} Fill NaN values of a Series.
\texttt{DataFrame.fillna} Fill NaN values of a DataFrame.
pandas.core.groupby.DataFrameGroupBy.corr

property DataFrameGroupBy.corr

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

- **method** [{'pearson', 'kendall', 'spearman'} or callable] Method of correlation:
  - pearson : standard correlation coefficient
  - kendall : Kendall Tau correlation coefficient
  - spearman : Spearman rank correlation
  - callable: callable with input two 1d ndarrays and returning a float. Note that the returned matrix from corr will have 1 along the diagonals and will be symmetric regardless of the callable’s behavior.

- **min_periods** [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for Pearson and Spearman correlation.

Returns

- **DataFrame** Correlation matrix.

See also:

- **DataFrame.corrwith** Compute pairwise correlation with another DataFrame or Series.
- **Series.corr** Compute the correlation between two Series.

Examples

```python
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
... >>> df = pd.DataFrame([(0.2, 0.3), (0.0, 0.6), (0.6, 0.0), (0.2, 0.1)],
...     columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
dogs    cats
dogs  1.00  0.30
cats  0.30  1.00
```

pandas.core.groupby.DataFrameGroupBy.count

DataFrameGroupBy.count()

Compute count of group, excluding missing values.

Returns

- **Series or DataFrame** Count of values within each group.

See also:

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.DataFrameGroupBy.cov

**property** DataFrameGroupBy.cov

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

**Parameters**

- **min_periods** [int, optional] Minimum number of observations required per pair of columns to have a valid result.
- **ddof** [int, default 1] Delta degrees of freedom. The divisor used in calculations is \(N - ddof\), where \(N\) represents the number of elements.

**Returns**

- **DataFrame** The covariance matrix of the series of the DataFrame.

**See also:**

- **Series.cov** Compute covariance with another Series.
- **core.window.ExponentialMovingWindow.cov** Exponential weighted sample covariance.
- **core.window.Expanding.cov** Expanding sample covariance.
- **core.window.Rolling.cov** Rolling sample covariance.

**Notes**

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-ddof.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

**Examples**

```python
def = pd.DataFrame([[1, 2], [0, 3], [2, 0], [1, 1]],
                    columns=['dogs', 'cats'])
def.cov()
```

```
   dogs  cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```
```python
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
   a      b     c      d      e
a  0.998438 -0.020161 0.059277 -0.008943 0.014144
b -0.020161 1.059352 -0.008543 -0.024738 0.009826
c  0.059277 -0.008543 1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486 0.921297 -0.013692
e  0.014144 0.009826 -0.000271 -0.013692 0.977795
```

### Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```python
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
   a       b    c
a 0.316741 NaN -0.150812
b  NaN 1.248003 0.191417
c -0.150812 0.191417 0.895202
```

---

**pandas.core.groupby.DataFrameGroupBy.cumcount**

`DataFrameGroupBy.cumcount(ascending=True)`

Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```python
self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))
```

**Parameters**

- `ascending` [bool, default True] If False, number in reverse, from length of group - 1 to 0.

**Returns**

- `Series` Sequence number of each element within each group.

**See also:**

- `ngroup` Number the groups themselves.
Examples

```python
>>> df = pd.DataFrame([['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
                    columns=['A'])
>>> df
  A
0 a
1 a
2 a
3 b
4 b
5 a

>>> df.groupby('A').cumcount()
   0  1  2  3  4  5
0  0  1  2  0  1  3
1  1  1  1  0  0  0
dtype: int64

>>> df.groupby('A').cumcount(ascending=False)
   0  1  2  3  4  5
0  3  2  1  0  0  0
1  1  1  0  0  0  0
dtype: int64
```

**pandas.core.groupby.DataFrameGroupBy.cummax**

DataFrameGroupBy.cummax(axis=0, **kwargs)

Cumulative max for each group.

Returns

Series or DataFrame

See also:

- Series.groupby: Apply a function groupby to a Series.
- DataFrame.groupby: Apply a function groupby to each row or column of a DataFrame.

**pandas.core.groupby.DataFrameGroupBy.cummin**

DataFrameGroupBy.cummin(axis=0, **kwargs)

Cumulative min for each group.

Returns

Series or DataFrame

See also:

- Series.groupby: Apply a function groupby to a Series.
- DataFrame.groupby: Apply a function groupby to each row or column of a DataFrame.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.core.groupby.DataFrameGroupBy.cumprod

DataFrameGroupBy.cumprod(axis=0, *args, **kwargs)
Cumulative product for each group.

Returns
Series or DataFrame

See also:
Series.groupby Apply a function groupby to a Series.
DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.DataFrameGroupBy.cumsum

DataFrameGroupBy.cumsum(axis=0, *args, **kwargs)
Cumulative sum for each group.

Returns
Series or DataFrame

See also:
Series.groupby Apply a function groupby to a Series.
DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.DataFrameGroupBy.describe

DataFrameGroupBy.describe(**kwargs)
Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is [.25, .5, .75], which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for Series. Here are the options:
- ‘all’: All columns of the input will be included in the output.
- A list-like of dtypes: Limits the results to the provided data types. To limit the result to numeric types submit numpy.number. To limit it instead to object columns submit the numpy.object data type. Strings can also be used in the style of select_dtypes (e.g. df.describe(include=['0'])). To select pandas categorical columns, use 'category'
- None (default): The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional] A black list of data types to omit from the result. Ignored for Series. Here are the options:
• A list-like of dtypes: Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use 'category'.

• None (default): The result will exclude nothing.

`datetime_is_numeric` [bool, default False] Whether to treat datetime dtypes as numeric. This affects statistics calculated for the column. For DataFrame input, this also controls whether datetime columns are included by default.

New in version 1.1.0.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

- `DataFrame.count` Count number of non-NA/null observations.
- `DataFrame.max` Maximum of the values in the object.
- `DataFrame.min` Minimum of the values in the object.
- `DataFrame.mean` Mean of the values.
- `DataFrame.std` Standard deviation of the observations.
- `DataFrame.select_dtypes` Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result’s index will include `count`, `mean`, `std`, `min`, `max` as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result’s index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value’s frequency. Timestamps also include the first and last items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a DataFrame, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a DataFrame are analyzed for the output. The parameters are ignored when analyzing a Series.

Examples

Describing a numeric Series.

```python
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count   3.0
mean  2.0
std   1.0
min   1.0
```

(continues on next page)
Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])
`print(s.describe())
count 4
unique 3
top a
top freq 2
dtype: object
```

Describing a timestamp Series.

```python
>>> s = pd.Series([... np.datetime64("2000-01-01"), ... np.datetime64("2010-01-01"), ... np.datetime64("2010-01-01") ... ])
`print(s.describe(datetime_is_numeric=True))
count 3
mean 2006-09-01 08:00:00
min 2000-01-01 00:00:00
25% 2004-12-31 12:00:00
50% 2010-01-01 00:00:00
75% 2010-01-01 00:00:00
max 2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```python
>>> df = pd.DataFrame({
"categorical": pd.Categorical(["d", "e", "f"],),
"numeric": [1, 2, 3],
"object": ["a", "b", "c"]
})
`print(df.describe())
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
```

Describing all columns of a DataFrame regardless of data type.

---

3.10. GroupBy
>>> df.describe(include='all')
categorical  numeric  object  
count  3  3.0  3
unique  3  NaN  3
top  f  NaN  a
freq  1  NaN  1
mean  NaN  2.0  NaN
std  NaN  1.0  NaN
min  NaN  1.0  NaN
25%  NaN  1.5  NaN
50%  NaN  2.0  NaN
75%  NaN  2.5  NaN
max  NaN  3.0  NaN

Describing a column from a DataFrame by accessing it as an attribute.

>>> df.numeric.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
Name: numeric, dtype: float64

Including only numeric columns in a DataFrame description.

>>> df.describe(include=[np.number])
numeric  
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0

Including only string columns in a DataFrame description.

>>> df.describe(include=[object])
object  
count 3
unique 3
top a
freq 1

Including only categorical columns from a DataFrame description.

>>> df.describe(include=['category'])
categorical  

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])
categorical  object
    count  3  3
    unique 3  3
    top    f  a
    freq   1  1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[object])
categorical  numeric
    count  3  3.0
    unique 3  NaN
    top    f  NaN
    freq   1  NaN
    mean   NaN  2.0
    std    NaN  1.0
    min    NaN  1.0
    25%    NaN  1.5
    50%    NaN  2.0
    75%    NaN  2.5
    max    NaN  3.0
```

### pandas.core.groupby.DataFrameGroupBy.diff

**property DataFrameGroupBy.diff**

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is element in previous row).

**Parameters**

- `periods` [int, default 1] Periods to shift for calculating difference, accepts negative values.
- `axis` [[0 or ‘index’, 1 or ‘columns’], default 0] Take difference over rows (0) or columns (1).

**Returns**

- `Dataframe` First differences of the Series.

**See also:**

- `Dataframe.pct_change` Percent change over given number of periods.
- `Dataframe.shift` Shift index by desired number of periods with an optional time freq.
- `Series.diff` First discrete difference of object.
Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in Dataframe, however dtype of the result is always float64.

Examples

Difference with previous row

```python
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
                    'b': [1, 1, 2, 3, 5, 8],
                    'c': [1, 4, 9, 16, 25, 36]})
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
<td>36</td>
</tr>
</tbody>
</table>
```

```python
>>> df.diff()
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>1.0</td>
<td>7.0</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
<td>2.0</td>
<td>9.0</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>3.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>
```

Difference with previous column

```python
>>> df.diff(axis=1)
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>-1</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>NaN</td>
<td>2</td>
<td>28</td>
</tr>
</tbody>
</table>
```

Difference with 3rd previous row

```python
>>> df.diff(periods=3)
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>2.0</td>
<td>15.0</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>4.0</td>
<td>21.0</td>
</tr>
<tr>
<td>5</td>
<td>3.0</td>
<td>6.0</td>
<td>27.0</td>
</tr>
</tbody>
</table>
```

Difference with following row
```python
>>> df.diff(periods=-1)
a  b  c
0 -1.0 0.0 -3.0
1 -1.0 -1.0 -5.0
2 -1.0 -1.0 -7.0
3 -1.0 -2.0 -9.0
4 -1.0 -3.0 -11.0
5 NaN NaN NaN
```

Overflow in input dtype

```python
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
a
0  NaN
1  255.0
```

### pandas.core.groupby.DataFrameGroupBy.ffill

**DataFrameGroupBy.ffill(limit=None)**
Forward fill the values.

**Parameters**

- `limit` [int, optional] Limit of how many values to fill.

**Returns**

- **Series or DataFrame** Object with missing values filled.

**See also:**

- `Series.ffill` Returns Series with minimum number of char in object.
- `DataFrame.ffill` Object with missing values filled or None if inplace=True.
- `Series.fillna` Fill NaN values of a Series.
- `DataFrame.fillna` Fill NaN values of a DataFrame.

### pandas.core.groupby.DataFrameGroupBy.fillna

**property** DataFrameGroupBy.fillna
Fill NA/Nan values using the specified method.

**Parameters**

- `value` [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.


- `axis` [{0 or ‘index’, 1 or ‘columns’}] Axis along which to fill missing values.

- `inplace` [bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).
**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

**downcast** [dict, default is None] A dict of item->dtype of what to downcast if possible, or the string ‘infer’ which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

**Returns**

*DataFrame or None* Object with missing values filled or None if inplace=True.

**See also:**

*interpolate* Fill NaN values using interpolation.

*reindex* Conform object to new index.

*asfreq* Convert TimeSeries to specified frequency.

**Examples**

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                      [3, 4, np.nan, 1],
                      [np.nan, np.nan, np.nan, np.nan],
                      [np.nan, 3, np.nan, 4]],
                     columns=list("ABCD"))
>>> df
   A  B     C  D
0  NaN 2.0  NaN  0.0
1  3.0 4.0  NaN  1.0
2  NaN NaN  NaN  NaN
3  NaN 3.0  NaN  4.0

Replace all NaN elements with 0s.

```python
>>> df.fillna(0)
   A  B  C  D
0 0.0 2.0 0.0 0.0
1 3.0 4.0 0.0 1.0
2 0.0 0.0 0.0 0.0
3 0.0 3.0 0.0 4.0
```

We can also propagate non-null values forward or backward.

```python
>>> df.fillna(method="ffill")
   A  B  C  D
0  NaN 2.0 NaN  0.0
1  3.0 4.0 0.0  1.0
2  0.0 0.0 0.0  0.0
3  0.0 3.0 0.0  4.0
```

Replace all NaN elements in column ‘A’, ‘B’, ‘C’, and ‘D’, with 0, 1, 2, and 3 respectively.

```python
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
```

(continues on next page)
A   B   C   D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0

Only replace the first NaN element.

>>> df.fillna(value=values, limit=1)
A   B   C   D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0

When filling using a DataFrame, replacement happens along the same column names and same indices

>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
A   B   C   D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0

Note that column D is not affected since it is not present in df2.

**pandas.core.groupby.DataFrameGroupBy.filter**

DataFrameGroupBy.filter(func, dropna=True, *args, **kwargs)

Return a copy of a DataFrame excluding filtered elements.

Elements from groups are filtered if they do not satisfy the boolean criterion specified by func.

**Parameters**

- **func** [function] Function to apply to each subframe. Should return True or False.
- **dropna** [Drop groups that do not pass the filter. True by default;] If False, groups that evaluate False are filled with NaNs.

**Returns**

- **filtered** [DataFrame]

**Notes**

Each subframe is endowed the attribute ‘name’ in case you need to know which group you are working on.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.
Examples

```python
>>> df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', ...
...                       'foo', 'bar'],
...                    'B': [1, 2, 3, 4, 5, 6],
...                    'C': [2.0, 5.0, 8.0, 1.0, 2.0, 9.0]})
>>> grouped = df.groupby('A')
>>> grouped.filter(lambda x: x['B'].mean() > 3.)
     A  B  C
   1 bar 2 5.0
   3 bar 4 1.0
   5 bar 6 9.0
```

### pandas.core.groupby.DataFrameGroupBy.hist

**property DataFrameGroupBy.hist**

Make a histogram of the DataFrame’s columns.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

**Parameters**

- `data` [DataFrame] The pandas object holding the data.
- `column` [str or sequence, optional] If passed, will be used to limit data to a subset of columns.
- `by` [object, optional] If passed, then used to form histograms for separate groups.
- `grid` [bool, default True] Whether to show axis grid lines.
- `xlabelsize` [int, default None] If specified changes the x-axis label size.
- `xrot` [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.
- `ylabelsize` [int, default None] If specified changes the y-axis label size.
- `yrot` [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.
- `ax` [Matplotlib axes object, default None] The axes to plot the histogram on.
- `sharex` [bool, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.
- `sharey` [bool, default False] In case subplots=True, share y axis and set some y axis labels to invisible.
- `figsize` [tuple, optional] The size in inches of the figure to create. Uses the value in `matplotlib.rcParams` by default.
- `layout` [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.
- `bins` [int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.
backend [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend. New in version 1.0.0.

legend [bool, default False] Whether to show the legend. New in version 1.1.0.

**kwargs All other plotting keyword arguments to be passed to matplotlib.pyplot.hist().

Returns
matplotlib.AxesSubplot or numpy.ndarray of them

See also:
matplotlib.pyplot.hist Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```python
>>> df = pd.DataFrame({
...    'length': [1.5, 0.5, 1.2, 0.9, 3],
...    'width': [0.7, 0.2, 0.15, 0.2, 1.1]
... }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```

**pandas.core.groupby.DataFrameGroupBy.idxmax**

DataFrameGroupBy.idxmax(axis=0, skipna=True)

Return index of first occurrence of maximum over requested axis.

NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise.

skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

Series Indexes of maxima along the specified axis.

Raises

ValueError

• If the row/column is empty

See also:

Series.idxmax Return index of the maximum element.
**Notes**

This method is the DataFrame version of `ndarray.argmax`.

**Examples**

Consider a dataset containing food consumption in Argentina.

```python
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48], ...
                   'co2_emissions': [37.2, 19.66, 1712]}, ...
                   index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
       consumption  co2_emissions  
Pork       10.51            37.20  
Wheat Products  103.11         19.66  
Beef         55.48          1712.00
```

By default, it returns the index for the maximum value in each column.

```python
>>> df.idxmax()
consumption    Wheat Products  
co2_emissions  Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis="columns"`.

```python
>>> df.idxmax(axis="columns")
Pork  co2_emissions  
Wheat Products  consumption  
Beef  co2_emissions  
dtype: object
```

**pandas.core.groupby.DataFrameGroupBy.idxmin**

`DataFrameGroupBy.idxmin(axis=0, skipna=True)`

Return index of first occurrence of minimum over requested axis.

NA/null values are excluded.

**Parameters**

- `axis` [0 or ‘index’, 1 or ‘columns’], default 0 The axis to use. 0 or ‘index’ for row-wise, 1 or ‘columns’ for column-wise.
- `skipna` [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

- `Series` Indexes of minima along the specified axis.

**Raises**

- `ValueError` If the row/column is empty
See also:

**Series.idxmin**  Return index of the minimum element.

**Notes**

This method is the DataFrame version of `ndarray.argmin`.

**Examples**

Consider a dataset containing food consumption in Argentina.

```python
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                     'co2_emissions': [37.2, 19.66, 1712]},
...                     index=['Pork', 'Wheat Products', 'Beef'])
```

```plaintext
   consumption  co2_emissions
Pork         10.51        37.20
Wheat Products 103.11       19.66
Beef          55.48       1712.00
```

By default, it returns the index for the minimum value in each column.

```python
>>> df.idxmin()
consumption     Pork
co2_emissions   Wheat Products
dtype: object
```

To return the index for the minimum value in each row, use `axis="columns"`.

```python
>>> df.idxmin(axis="columns")
Pork    consumption
Wheat Products  co2_emissions
Beef    consumption
dtype: object
```

**pandas.core.groupby.DataFrameGroupBy.mad**

**property**  DataFrameGroupBy.mad

Return the mean absolute deviation of the values over the requested axis.

**Parameters**

- **axis**  [[index (0), columns (1)]]  Axis for the function to be applied on.
- **skipna**  [bool, default True]  Exclude NA/null values when computing the result.
- **level**  [int or level name, default None]  If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**Returns**

Series or DataFrame (if level specified)
DataFrameGroupBy.nunique(dropna=True)

Return DataFrame with counts of unique elements in each position.

Parameters

- dropna [bool, default True] Don’t include NaN in the counts.

Returns

- DataFrame

Examples

```python
>>> df = pd.DataFrame({'id': ['spam', 'egg', 'egg', 'spam',
  'ham', 'ham'],
  'value1': [1, 5, 5, 2, 5, 5],
  'value2': list('abbaxy')})

>>> df
    id  value1 value2
  0  spam     1     a
  1   egg     5     b
  2   egg     5     b
  3  spam     2     a
  4   ham     5     x
  5   ham     5     y

>>> df.groupby('id').nunique()
    value1 value2
id
egg   1     1
ham   1     2
spam  2     1

Check for rows with the same id but conflicting values:

```python
>>> df.groupby('id').filter(lambda g: (g.nunique() > 1).any())
    id  value1 value2
  0  spam     1     a
  3  spam     2     a
  4   ham     5     x
  5   ham     5     y
```

DataFrameGroupBy.pad(limit=None)

Forward fill the values.

Parameters

- limit [int, optional] Limit of how many values to fill.

Returns

- Series or DataFrame Object with missing values filled.
See also:

- **Series.fill** Returns Series with minimum number of char in object.
- **DataFrame.fill** Object with missing values filled or None if inplace=True.
- **Series.fillna** Fill NaN values of a Series.
- **DataFrame.fillna** Fill NaN values of a DataFrame.

### pandas.core.groupby.DataFrameGroupBy.pct_change

DataFrameGroupBy.pct_change(periods=1, fill_method='ffill', limit=None, freq=None, axis=0)

Calculate pct_change of each value to previous entry in group.

#### Returns

- **Series or DataFrame** Percentage changes within each group.

See also:

- **Series.groupby** Apply a function groupby to a Series.
- **DataFrame.groupby** Apply a function groupby to each row or column of a DataFrame.

### pandas.core.groupby.DataFrameGroupBy.plot

**property** DataFrameGroupBy.plot

Class implementing the .plot attribute for groupby objects.

### pandas.core.groupby.DataFrameGroupBy.quantile

DataFrameGroupBy.quantile(q=0.5, interpolation='linear')

Return group values at the given quantile, a la numpy.percentile.

#### Parameters

- **q** [float or array-like, default 0.5 (50% quantile)] Value(s) between 0 and 1 providing the quantile(s) to compute.

- **interpolation** [[‘linear’, ‘lower’, ‘higher’, ‘midpoint’, ‘nearest’]] Method to use when the desired quantile falls between two points.

#### Returns

- **Series or DataFrame** Return type determined by caller of GroupBy object.

See also:

- **Series.quantile** Similar method for Series.
- **DataFrame.quantile** Similar method for DataFrame.
- **numpy.percentile** NumPy method to compute qth percentile.
Examples

```python
>>> df = pd.DataFrame([  
...     ['a', 1], ['a', 2], ['a', 3],  
...     ['b', 1], ['b', 3], ['b', 5]  
... ], columns=['key', 'val'])
>>> df.groupby('key').quantile()
      val
key  
  a  2.0
  b  3.0
```

**pandas.core.groupby.DataFrameGroupBy.rank**

`DataFrameGroupBy.rank(method='average', ascending=True, na_option='keep', pct=False, axis=0)`

Provide the rank of values within each group.

**Parameters**

  - average: average rank of group.
  - min: lowest rank in group.
  - max: highest rank in group.
  - first: ranks assigned in order they appear in the array.
  - dense: like ‘min’, but rank always increases by 1 between groups.
- `ascending` [bool, default True] False for ranks by high (1) to low (N).
- `na_option` [{‘keep’, ‘top’, ‘bottom’}, default ‘keep’]
  - keep: leave NA values where they are.
  - top: smallest rank if ascending.
  - bottom: smallest rank if descending.
- `pct` [bool, default False] Compute percentage rank of data within each group.
- `axis` [int, default 0] The axis of the object over which to compute the rank.

**Returns**

- `DataFrame with ranking of values within each group`

**See also:**

- `Series.groupby` Apply a function groupby to a Series.
- `DataFrame.groupby` Apply a function groupby to each row or column of a DataFrame.
Examples

```python
>>> df = pd.DataFrame(
    ... {
    ...     "group": ["a", "a", "a", "a", "b", "b", "b", "b"],
    ...     "value": [2, 4, 2, 3, 1, 2, 4, 1],
    ... } )
>>> df
   group  value
0    a     2
1    a     4
2    a     2
3    a     3
4    a     5
5    b     1
6    b     2
7    b     4
8    b     1
9    b     5
```

```python
>>> for method in ['average', 'min', 'max', 'dense', 'first']:
    ...     df[f'value_{method}_rank'] = df.groupby('group')['value'].rank(method)
>>> df
   group  value  average_rank  min_rank  max_rank  dense_rank  first_rank
0    a     2        1.5        1.0    2.0        1.0    1.0
1    a     4        4.0        4.0    4.0        3.0    4.0
2    a     2        1.5        1.0    2.0        1.0    2.0
3    a     3        3.0        3.0    3.0        2.0    3.0
4    a     5        5.0        5.0    5.0        4.0    5.0
5    b     1        1.5        1.0    2.0        1.0    1.0
6    b     2        3.0        3.0    3.0        2.0    3.0
7    b     4        4.0        4.0    4.0        3.0    4.0
8    b     1        1.5        1.0    2.0        1.0    2.0
9    b     5        5.0        5.0    5.0        4.0    5.0
```

### pandas.core.groupby.DataFrameGroupBy.resample

DataFrameGroupBy.resample(*rule*, *args*, **kwargs)

Provide resampling when using a TimeGrouper.

Given a grouper, the function resamples it according to a string “string” -> “frequency”.

See the frequency aliases documentation for more details.

**Parameters**

- **rule** [str or DateOffset] The offset string or object representing target grouper conversion.
- **args**, **kwargs** Possible arguments are how, fill_method, limit, kind and on, and other arguments of TimeGrouper.

**Returns**

- **Grouper** Return a new grouper with our resampler appended.

See also:

- **Grouper** Specify a frequency to resample with when grouping by a key.
**DatetimeIndex.resample** Frequency conversion and resampling of time series.

### Examples

```python
>>> idx = pd.date_range('1/1/2000', periods=4, freq='T')

>>> df = pd.DataFrame(data=4 * range(2),
                    index=idx,
                    columns=['a', 'b'])

>>> df.iloc[2, 0] = 5

>>> df
   a  b
2000-01-01 00:00:00 0 1
2000-01-01 00:01:00 0 1
2000-01-01 00:02:00 5 1
2000-01-01 00:03:00 0 1

Downsample the DataFrame into 3 minute bins and sum the values of the timestamps falling into a bin.

```python
>>> df.groupby('a').resample('3T').sum()
   a  b
   a
     0 2000-01-01 00:00:00 0 2
     2000-01-01 00:03:00 0 1
     5 2000-01-01 00:00:00 5 1
```

Upsample the series into 30 second bins.

```python
>>> df.groupby('a').resample('30S').sum()
   a  b
   a
     0 2000-01-01 00:00:00 0 1
     2000-01-01 00:00:30 0 0
     2000-01-01 00:01:00 0 1
     2000-01-01 00:01:30 0 0
     2000-01-01 00:02:00 0 0
     2000-01-01 00:02:30 0 0
     2000-01-01 00:03:00 0 1
     5 2000-01-01 00:02:00 5 1
```

Resample by month. Values are assigned to the month of the period.

```python
>>> df.groupby('a').resample('M').sum()
   a  b
   a
     0 2000-01-31 0 3
     5 2000-01-31 5 1
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```python
>>> df.groupby('a').resample('3T', closed='right').sum()
   a  b
   a
     0 1999-12-31 23:57:00 0 1
```
Downsample the series into 3 minute bins and close the right side of the bin interval, but label each bin using the right edge instead of the left.

```python
>>> df.groupby('a').resample('3T', closed='right', label='right').sum()
   a   b
0  2000-01-01 00:00:00  0  1
   2000-01-01 00:03:00  0  2
5  2000-01-01 00:03:00  5  1
```

**pandas.core.groupby.DataFrameGroupBy.sample**

`DataFrameGroupBy.sample(n=None, frac=None, replace=False, weights=None, random_state=None)`

Return a random sample of items from each group.

You can use `random_state` for reproducibility.

New in version 1.1.0.

**Parameters**

- `n` [int, optional] Number of items to return for each group. Cannot be used with `frac` and must be no larger than the smallest group unless `replace` is True. Default is one if `frac` is None.
- `frac` [float, optional] Fraction of items to return. Cannot be used with `n`.
- `replace` [bool, default False] Allow or disallow sampling of the same row more than once.
- `weights` [list-like, optional] Default None results in equal probability weighting. If passed a list-like then values must have the same length as the underlying DataFrame or Series object and will be used as sampling probabilities after normalization within each group. Values must be non-negative with at least one positive element within each group.
- `random_state` [int, array-like, BitGenerator, np.random.RandomState, np.random.Generator, optional] If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given. Changed in version 1.4.0: np.random.Generator objects now accepted

**Returns**

- `Series` or `DataFrame` A new object of same type as caller containing items randomly sampled within each group from the caller object.

**See also:**

- `DataFrame.sample` Generate random samples from a DataFrame object.
- `numpy.random.choice` Generate a random sample from a given 1-D numpy array.
Examples

```python
>>> df = pd.DataFrame(
... {'a': ['red'] * 2 + ['blue'] * 2 + ['black'] * 2, 'b': range(6)}
... )
>>> df
   a b
0 red 0
1 red 1
2 blue 2
3 blue 3
4 black 4
5 black 5

Select one row at random for each distinct value in column a. The `random_state` argument can be used to guarantee reproducibility:

```python
>>> df.groupby('a').sample(n=1, random_state=1)
   a b
4 black 4
2 blue 2
1 red 1
```

Set `frac` to sample fixed proportions rather than counts:

```python
>>> df.groupby('a')['b'].sample(frac=0.5, random_state=2)
5 5
2 2
0 0
Name: b, dtype: int64
```

Control sample probabilities within groups by setting weights:

```python
>>> df.groupby('a').sample(
... n=1,
... weights=[1, 1, 1, 0, 0, 1],
... random_state=1,
... )
   a b
5 black 5
2 blue 2
0 red 0
```

**pandas.core.groupby.DataFrameGroupBy.shift**

DataFrameGroupBy.shift(periods=1, freq=None, axis=0, fill_value=None)

Shift each group by periods observations.

If freq is passed, the index will be increased using the periods and the freq.

**Parameters**

- **periods** [int, default 1] Number of periods to shift.
- **freq** [str, optional] Frequency string.
axis [axis to shift, default 0] Shift direction.
fill_value [optional] The scalar value to use for newly introduced missing values.

Returns

Series or DataFrame Object shifted within each group.

See also:

Index.shift Shift values of Index.
tshift Shift the time index, using the index’s frequency if available.

pandas.core.groupby.DataFrameGroupBy.size

DataFrameGroupBy.size() Compute group sizes.

Returns

DataFrame or Series Number of rows in each group as a Series if as_index is True or a
DataFrame if as_index is False.

See also:

Series.groupby Apply a function groupby to a Series.
DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

pandas.core.groupby.DataFrameGroupBy.skew

property DataFrameGroupBy.skew

Return unbiased skew over requested axis.

Normalized by N-1.

Parameters

axis [[index (0), columns (1)]] Axis for the function to be applied on.
skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along
a particular level, collapsing into a Series.
numeric_only [bool, default None] Include only float, int, boolean columns. If None, will
attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)
pandas.core.groupby.DataFrameGroupBy.take

**property** DataFrameGroupBy.take

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters**

- **indices** [array-like] An array of ints indicating which positions to take.
- **axis** [0 or ‘index’, 1 or ‘columns’, None], default 0] The axis on which to select elements.
- **is_copy** [bool] Before pandas 1.0, is_copy=False can be specified to ensure that the return value is an actual copy. Starting with pandas 1.0, take always returns a copy, and the keyword is therefore deprecated.

Deprecated since version 1.0.0.

**kwargs** For compatibility with numpy.take(). Has no effect on the output.

**Returns**

- **taken** [same type as caller] An array-like containing the elements taken from the object.

**See also:**

- **DataFrame.loc** Select a subset of a DataFrame by labels.
- **DataFrame.iloc** Select a subset of a DataFrame by positions.
- **numpy.take** Take elements from an array along an axis.

**Examples**

```python
>>> df = pd.DataFrame([(
    'falcon', 'bird', 389.0),
    ('parrot', 'bird', 24.0),
    ('lion', 'mammal', 80.5),
    ('monkey', 'mammal', np.nan)],
   columns=['name', 'class', 'max_speed'],
   index=[0, 2, 3, 1])
>>> df
name    class  max_speed
0  falcon   bird     389.0
2  parrot   bird      24.0
3   lion   mammal     80.5
1  monkey   mammal      NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That’s because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```python
>>> df.take([0, 3])
name    class  max_speed
0  falcon   bird     389.0
1  monkey   mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).
We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```python
>>> df.take([-1, -2])
  name  class  max_speed
1  monkey  mammal  NaN
3   lion  mammal   80.5
```

### pandas.core.groupby.DataFrameGroupBy.tshift

#### property DataFrameGroupBy.tshift

Shift the time index, using the index’s frequency if available.

Deprecated since version 1.1.0: Use shift instead.

**Parameters**

- **periods** [int] Number of periods to move, can be positive or negative.
- **freq** [DateOffset, timedelta, or str, default None] Increment to use from the tseries module or time rule expressed as a string (e.g. ‘EOM’).
- **axis** [0 or ‘index’, 1 or ‘columns’, None, default 0] Corresponds to the axis that contains the Index.

**Returns**

- **shifted** [Series/DataFrame]

**Notes**

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

### pandas.core.groupby.DataFrameGroupBy.value_counts

DataFrameGroupBy.value_counts(subset=None, normalize=False, sort=True, ascending=False, dropna=True)

Return a Series or DataFrame containing counts of unique rows.

New in version 1.4.0.

**Parameters**

- **subset** [list-like, optional] Columns to use when counting unique combinations.
- **normalize** [bool, default False] Return proportions rather than frequencies.
- **sort** [bool, default True] Sort by frequencies.
- **ascending** [bool, default False] Sort in ascending order.
- **dropna** [bool, default True] Don’t include counts of rows that contain NA values.
Returns

Series or DataFrame Series if the groupby as_index is True, otherwise DataFrame.

See also:

Series.value_counts Equivalent method on Series.
DataFrame.value_counts Equivalent method on DataFrame.
SeriesGroupBy.value_counts Equivalent method on SeriesGroupBy.

Notes

• If the groupby as_index is True then the returned Series will have a MultiIndex with one level per input column.
• If the groupby as_index is False then the returned DataFrame will have an additional column with the value_counts. The column is labelled ‘count’ or ‘proportion’, depending on the normalize parameter.

By default, rows that contain any NA values are omitted from the result.

By default, the result will be in descending order so that the first element of each group is the most frequently-occurring row.

Examples

```python
>>> df = pd.DataFrame({
...     'gender': ['male', 'male', 'female', 'male', 'female', 'male'],
...     'education': ['low', 'medium', 'high', 'low', 'high', 'low'],
...     'country': ['US', 'FR', 'US', 'FR', 'FR', 'FR']
... })
```

```python
>>> df
    gender  education  country
0     male       low      US
1     male  medium      FR
2   female     high      US
3     male       low      FR
4   female     high      FR
5     male       low      FR
```

```python
>>> df.groupby('gender').value_counts()

<table>
<thead>
<tr>
<th>gender</th>
<th>education</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>female</td>
<td>high</td>
<td>FR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>US</td>
</tr>
<tr>
<td>male</td>
<td>low</td>
<td>FR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>FR</td>
</tr>
</tbody>
</table>

dtype: int64
```

```python
>>> df.groupby('gender').value_counts(ascending=True)

<table>
<thead>
<tr>
<th>gender</th>
<th>education</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>female</td>
<td>high</td>
<td>FR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>US</td>
</tr>
<tr>
<td>male</td>
<td>low</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>FR</td>
</tr>
</tbody>
</table>
```

(continues on next page)
The following methods are available only for SeriesGroupBy objects.

- **SeriesGroupBy.hist**
  - Draw histogram of the input series using matplotlib.
- **SeriesGroupBy.nlargest([n, keep])**
  - Return the largest n elements.
- **SeriesGroupBy.nsmallest([n, keep])**
  - Return the smallest n elements.
- **SeriesGroupBy.nunique([dropna])**
  - Return number of unique elements in the group.
- **SeriesGroupBy.unique**
  - Return unique values of Series object.
- **SeriesGroupBy.value_counts([normalize, ...])**
  - Alias for is_monotonic.
- **SeriesGroupBy.is_monotonic_increasing**
  - Return boolean if values in the object are monotonic_increasing.
- **SeriesGroupBy.is_monotonic_decreasing**
  - Return boolean if values in the object are monotonic_decreasing.
pandas.core.groupby.SeriesGroupBy.hist

property SeriesGroupBy.hist

Draw histogram of the input series using matplotlib.

Parameters

by [object, optional] If passed, then used to form histograms for separate groups.
ax [matplotlib axis object] If not passed, uses gca().
grid [bool, default True] Whether to show axis grid lines.
xlabelsize [int, default None] If specified changes the x-axis label size.
xrot [float, default None] Rotation of x axis labels.
ylabelsize [int, default None] If specified changes the y-axis label size.
yrot [float, default None] Rotation of y axis labels.
figsize [tuple, default None] Figure size in inches by default.

bins [int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend.

legend [bool, default False] Whether to show the legend.

**kwargs To be passed to the actual plotting function.

Returns

matplotlib.AxesSubplot A histogram plot.

See also:

matplotlib.axes.Axes.hist Plot a histogram using matplotlib.

pandas.core.groupby.SeriesGroupBy.nlargest

SeriesGroupBy.nlargest(n=5, keep=’first’)

Return the largest n elements.

Parameters

n [int, default 5] Return this many descending sorted values.
keep [('first', 'last', 'all'), default 'first'] When there are duplicate values that cannot all fit in a Series of n elements:

• first: return the first n occurrences in order of appearance.
• last: return the last n occurrences in reverse order of appearance.
• all: keep all occurrences. This can result in a Series of size larger than n.

Returns
Series The n largest values in the Series, sorted in decreasing order.

See also:
Series.nsmallest Get the n smallest elements.
Series.sort_values Sort Series by values.
Series.head Return the first n rows.

Notes

Faster than .sort_values(ascending=False).head(n) for small n relative to the size of the Series object.

Examples

```python
>>> countries_population = {
  "Italy": 59000000,
  "France": 65000000,
  ...
  "Malta": 4344000,
  "Maldives": 4344000,
  ...
  "Brunei": 4344000,
  "Iceland": 337000,
  ...
  "Nauru": 11300,
  "Tuvalu": 11300,
  ...
  "Anguilla": 11300,
  "Montserrat": 5200
}

>>> s = pd.Series(countries_population)

>>> s
Italy  59000000
France  65000000
Malta   4344000
Maldives  4344000
Brunei   4344000
Iceland  337000
Nauru    11300
Tuvalu   11300
Anguilla 11300
Montserrat  5200
dtype: int64

The n largest elements where n=5 by default.

```python
>>> s.nlargest()
France  65000000
Italy  59000000
Malta  4344000
Maldives  4344000
Brunei  4344000
dtype: int64

The n largest elements where n=3. Default keep value is ‘first’ so Malta will be kept.

```python
>>> s.nlargest(3)
France  65000000
Italy  59000000
Malta  4344000
Maldives  4344000
Brunei  4344000
dtype: int64

The n largest elements where n=3 and keeping the last duplicates. Brunei will be kept since it is the last with value 4344000 based on the index order.
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> s.nlargest(3, keep='last')
France    65000000
Italy      59000000
Brunei      434000
 dtype: int64
```

The $n$ largest elements where $n$=3 with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```python
>>> s.nlargest(3, keep='all')
France    65000000
Italy      59000000
Malta       434000
Maldives   434000
Brunei      434000
dtype: int64
```

**pandas.core.groupby.SeriesGroupBy.nsmallest**

`SeriesGroupBy.nsmallest(n=5, keep='first')`

Return the smallest $n$ elements.

**Parameters**

- **n** [int, default 5] Return this many ascending sorted values.
- **keep** [‘first’, ‘last’, ‘all’], default ‘first’] When there are duplicate values that cannot all fit in a Series of $n$ elements:
  - **first** : return the first $n$ occurrences in order of appearance.
  - **last** : return the last $n$ occurrences in reverse order of appearance.
  - **all** : keep all occurrences. This can result in a Series of size larger than $n$.

**Returns**

`Series` The $n$ smallest values in the Series, sorted in increasing order.

**See also:**

- `Series.nlargest` Get the $n$ largest elements.
- `Series.sort_values` Sort Series by values.
- `Series.head` Return the first $n$ rows.

**Notes**

Faster than `.sort_values().head(n)` for small $n$ relative to the size of the Series object.
Examples

```python
>>> countries_population = {
    "Italy": 59000000, "France": 65000000, 
    "Brunei": 434000, "Malta": 434000, 
    "Maldives": 434000, "Iceland": 337000, 
    "Nauru": 11300, "Tuvalu": 11300, 
    "Anguilla": 11300, "Montserrat": 5200
}

>>> s = pd.Series(countries_population)

>>> s
Italy     59000000
France    65000000
Brunei    434000
Malta     434000
Maldives  434000
Iceland   337000
Nauru     11300
Tuvalu    11300
Anguilla  11300
Montserrat 5200
dtype: int64

The n smallest elements where n=5 by default.

```python
default
Montserrat  5200
Nauru       11300
Tuvalu      11300
Anguilla    11300
Iceland     337000
dtype: int64
```

The n smallest elements where n=3. Default keep value is `first` so Nauru and Tuvalu will be kept.

```python
>>> s.nsmallest(3)
Montserrat  5200
Nauru       11300
Tuvalu      11300
dtype: int64
```

The n smallest elements where n=3 and keeping the last duplicates. Anguilla and Tuvalu will be kept since they are the last with value 11300 based on the index order.

```python
>>> s.nsmallest(3, keep='last')
Montserrat  5200
Anguilla    11300
Tuvalu      11300
dtype: int64
```

The n smallest elements where n=3 with all duplicates kept. Note that the returned Series has four elements due to the three duplicates.

```python
>>> s.nsmallest(3, keep='all')
Montserrat  5200
Nauru       11300
```
Tuvalu  11300
Anguilla  11300
dtype: int64

pandas.core.groupby.SeriesGroupBy.nunique

SeriesGroupBy.nunique(dropna=True)

Return number of unique elements in the group.
Returns
Series  Number of unique values within each group.

pandas.core.groupby.SeriesGroupBy.unique

property SeriesGroupBy.unique

Return unique values of Series object.
Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.
Returns
ndarray or ExtensionArray  The unique values returned as a NumPy array. See Notes.
See also:
unique  Top-level unique method for any 1-d array-like object.
Index.unique  Return Index with unique values from an Index object.

Notes

Returns the unique values as a NumPy array. In case of an extension-array backed Series, a new
ExtensionArray of that type with just the unique values is returned. This includes
- Categorical
- Period
- Datetime with Timezone
- Interval
- Sparse
- IntegerNA
See Examples section.

Examples

```python
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```python
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```python
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
... for _ in range(3)]).unique()
<DatetimeArray>
```

An Categorical will return categories in the order of appearance and with the same dtype.

```python
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
['b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'), ordered=True)).unique()
['b', 'a', 'c']
Categories (3, object): ['a' < 'b' < 'c']
```

### pandas.core.groupby.SeriesGroupBy.value_counts

```python
SeriesGroupBy.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)
```

### pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing

**property** SeriesGroupBy.is_monotonic_increasing

Alias for is_monotonic.

### pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing

**property** SeriesGroupBy.is_monotonic_decreasing

Return boolean if values in the object are monotonic_decreasing.

**Returns**

bool

The following methods are available only for DataFrameGroupBy objects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrameGroupBy.corrwith</td>
<td>Compute pairwise correlation.</td>
</tr>
<tr>
<td>DataFrameGroupBy.boxplot</td>
<td>Make box plots from DataFrameGroupBy data.</td>
</tr>
</tbody>
</table>

### pandas.core.groupby.DataFrameGroupBy.corrwith

**property** DataFrameGroupBy.corrwith

Compute pairwise correlation.

Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

**Parameters**

- **other** [DataFrame, Series] Object with which to compute correlations.
- **axis** [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' to compute column-wise, 1 or 'columns' for row-wise.
- **drop** [bool, default False] Drop missing indices from result.
- **method** [{‘pearson’, ‘kendall’, ‘spearman’} or callable] Method of correlation:
• pearson : standard correlation coefficient
• kendall : Kendall Tau correlation coefficient
• spearman : Spearman rank correlation
• callable: callable with input two 1d ndarrays and returning a float.

Returns

Series  Pairwise correlations.

See also:

DataFrame.corr  Compute pairwise correlation of columns.

pandas.core.groupby.DataFrameGroupBy.boxplot

DataFrameGroupBy.boxplot(subplots=True, column=None, fontsize=None, rot=0, grid=True, ax=None, figsize=None, layout=None, sharex=False, sharey=True, backend=None, **kwargs)

Make box plots from DataFrameGroupBy data.

Parameters

grouped  [Grouped DataFrame]
subplots  [bool]
    • False - no subplots will be used
    • True - create a subplot for each group.
column  [column name or list of names, or vector] Can be any valid input to groupby.
fontsize  [int or str]
rot  [label rotation angle]
grid  [Setting this to True will show the grid]
ax  [Matplotlib axis object, default None]
figsize  [A tuple (width, height) in inches]
layout  [tuple (optional)] The layout of the plot: (rows, columns).
sharex  [bool, default False] Whether x-axes will be shared among subplots.
sharey  [bool, default True] Whether y-axes will be shared among subplots.
backend  [str, default None] Backend to use instead of the backend specified in the option plotting.backend. For instance, ‘matplotlib’. Alternatively, to specify the plotting.backend for the whole session, set pd.options.plotting.backend.

New in version 1.0.0.

**kwargs  All other plotting keyword arguments to be passed to matplotlib’s boxplot function.

Returns

dict of key/value = group key/DataFrame.boxplot return value

or DataFrame.boxplot return value in case subplots=figures=False
Examples

You can create boxplots for grouped data and show them as separate subplots:

```python
>>> import itertools
>>> tuples = [t for t in itertools.product(range(1000), range(4))]
>>> index = pd.MultiIndex.from_tuples(tuples, names=['lvl0', 'lvl1'])
>>> data = np.random.randn(len(index),4)
>>> df = pd.DataFrame(data, columns=list('ABCD'), index=index)
>>> grouped = df.groupby(level='lvl1')
>>> grouped.boxplot(rot=45, fontsize=12, figsize=(8,10))
```

The subplots=False option shows the boxplots in a single figure:

```python
>>> grouped.boxplot(subplots=False, rot=45, fontsize=12)
```

3.11 Resampling

Resampler objects are returned by resample calls: `pandas.DataFrame.resample()`, `pandas.Series.resample()`.

3.11.1 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Resampler.__iter__()</code></td>
<td>Groupby iterator.</td>
</tr>
<tr>
<td><code>Resampler.groups</code></td>
<td>Dict {group name -&gt; group labels}.</td>
</tr>
<tr>
<td><code>Resampler.indices</code></td>
<td>Dict {group name -&gt; group indices}.</td>
</tr>
<tr>
<td><code>Resampler.get_group(name[, obj])</code></td>
<td>Construct DataFrame from group with provided name.</td>
</tr>
</tbody>
</table>

**pandas.core.resample.Resampler.__iter__**

Resampler.__iter__()

Groupby iterator.

**Returns**

Generator yielding sequence of (name, subsetted object)

for each group

**pandas.core.resample.Resampler.groups**

**property** Resampler.groups

Dict {group name -> group labels}. 
3.11. Resampling
pandas.core.resample.Resampler.indices

**property** Resampler.indices

Dict {group name -> group indices}.

pandas.core.resample.Resampler.get_group

Resampler.get_group(name, obj=None)

Construct DataFrame from group with provided name.

**Parameters**

- name [object] The name of the group to get as a DataFrame.
- obj [DataFrame, default None] The DataFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used.

**Returns**

- group [same type as obj]

### 3.11.2 Function application

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Resampler.apply([func])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>Resampler.aggregate([func])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>Resampler.transform(arg, *args, **kwargs)</code></td>
<td>Call function producing a like-indexed Series on each group and return a Series with the transformed values.</td>
</tr>
<tr>
<td><code>Resampler.pipe(func, *args, **kwargs)</code></td>
<td>Apply a function <code>func</code> with arguments to this Resampler object and return the function's result.</td>
</tr>
</tbody>
</table>

pandas.core.resample.Resampler.apply

`Resampler.apply(func=none, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

**Parameters**

- `func` [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

  Accepted combinations are:
  
  - function
  - string function name
  - list of functions and/or function names, e.g. `[np.sum, 'mean']`
  - dict of axis labels -> functions, function names or list of such.

- `*args` Positional arguments to pass to `func`.

- `**kwargs` Keyword arguments to pass to `func`.

**Returns**

- scalar, Series or DataFrame The return can be:
• scalar : when Series.agg is called with single function
• Series : when DataFrame.agg is called with a single function
• DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:
DataFrame.groupby.aggregate Aggregate using callable, string, dict, or list of string/callables.
DataFrame.resample.transform Transforms the Series on each group based on the given function.
DataFrame.aggregate Aggregate using one or more operations over the specified axis.

Notes

agg is an alias for aggregate. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
>>> s = pd.Series([1, 2, 3, 4, 5],
                index=pd.date_range('20130101', periods=5, freq='S'))
```

```python
>>> s
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
2013-01-01 00:00:04    5
Freq: S, dtype: int64
```

```python
>>> r = s.resample('2S')
```

```python
>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64
```

```python
>>> r.agg(['sum', 'mean', 'max'])
                      sum  mean  max
2013-01-01 00:00:00    3    1.5    2
2013-01-01 00:00:02    7    3.5    4
2013-01-01 00:00:04    5    5.0    5
```

```python
>>> r.agg({'result': lambda x: x.mean() / x.std(),
         'total': np.sum})
result  total
2013-01-01 00:00:00  2.121320    3
2013-01-01 00:00:02  4.949747    7
2013-01-01 00:00:04  NaN    5
```
```python
>>> r.agg(average=\"mean\", total=\"sum\")

<table>
<thead>
<tr>
<th></th>
<th>average</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00</td>
<td>1.5</td>
<td>3</td>
</tr>
<tr>
<td>2013-01-01 00:00:02</td>
<td>3.5</td>
<td>7</td>
</tr>
<tr>
<td>2013-01-01 00:00:04</td>
<td>5.0</td>
<td>5</td>
</tr>
</tbody>
</table>
```

**pandas.core.resample.Resampler.aggregate**

```python
Resampler.aggregate(func=None, *args, **kwargs)
```

Aggregate using one or more operations over the specified axis.

**Parameters**

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.
  
  Accepted combinations are:
  
  - function
  - string function name
  - list of functions and/or function names, e.g. [np.sum, \'mean\']
  - dict of axis labels -> functions, function names or list of such.

- **args** Positional arguments to pass to `func`.

- **kwargs** Keyword arguments to pass to `func`.

**Returns**

- scalar, Series or DataFrame The return can be:
  
  - scalar : when Series.agg is called with single function
  - Series : when DataFrame.agg is called with a single function
  - DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

**See also:**

- `DataFrame.groupby.aggregate` Aggregate using callable, string, dict, or list of string/callables.
- `DataFrame.resample.transform` Transforms the Series on each group based on the given function.
- `DataFrame.aggregate` Aggregate using one or more operations over the specified axis.

**Notes**

- `agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See *Mutating with User Defined Function (UDF) methods* for more details.

A passed user-defined-function will be passed a Series for evaluation.
Examples

```python
>>> s = pd.Series([1, 2, 3, 4, 5],
                 index=pd.date_range('20130101', periods=5, freq='S'))
>>> s
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
2013-01-01 00:00:04    5
Freq: S, dtype: int64

>>> r = s.resample('2S')

>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64

>>> r.agg(['sum', 'mean', 'max'])
        sum  mean  max
2013-01-01 00:00:00    3  1.5   2
2013-01-01 00:00:02    7  3.5   4
2013-01-01 00:00:04    5  5.0   5

>>> r.agg({'result': lambda x: x.mean() / x.std(),
         'total': np.sum})
      result  total
2013-01-01 00:00:00 2.121320    3
2013-01-01 00:00:02 4.949747    7
2013-01-01 00:00:04  NaN      5

>>> r.agg(average="mean", total="sum")
       average  total
2013-01-01 00:00:00   1.5     3
2013-01-01 00:00:02   3.5     7
2013-01-01 00:00:04   5.0     5
```

**pandas.core.resample.Resampler.transform**

Resampler.transform(arg, *args, **kwargs)

Call function producing a like-indexed Series on each group and return a Series with the transformed values.

Parameters

- **arg** [function] To apply to each group. Should return a Series with the same index.

Returns

- **transformed** [Series]
Examples

```python
>>> s = pd.Series([1, 2],
...                index=pd.date_range('20180101',
...                                    periods=2,
...                                    freq='1h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
Freq: H, dtype: int64

>>> resampled = s.resample('15min')
>>> resampled.transform(lambda x: (x - x.mean()) / x.std())
2018-01-01 00:00:00    NaN
2018-01-01 01:00:00    NaN
Freq: H, dtype: float64
```

**pandas.core.resample.Resampler.pipe**

`Resampler.pipe(func, *args, **kwargs)`

Apply a function `func` with arguments to this Resampler object and return the function’s result.

Use `.pipe` when you want to improve readability by chaining together functions that expect Series, DataFrames, GroupBy or Resampler objects. Instead of writing

```python
>>> h(g(f(df.groupby('group'))), arg1=a, arg2=b, arg3=c)
```

You can write

```python
>>> (df.groupby('group')
...    .pipe(f)
...    .pipe(g, arg1=a)
...    .pipe(h, arg2=b, arg3=c))
```

which is much more readable.

**Parameters**

- `func` [callable or tuple of (callable, str)] Function to apply to this Resampler object or, alternatively, a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Resampler object.

- `args` [iterable, optional] Positional arguments passed into `func`.


**Returns**

- `object` [the return type of `func`.

**See also:**

- `Series.pipe` Apply a function with arguments to a series.
- `DataFrame.pipe` Apply a function with arguments to a dataframe.
- `apply` Apply function to each group instead of to the full Resampler object.
Notes

See more here

Examples

```python
>>> df = pd.DataFrame({'A': [1, 2, 3, 4]},
                    index=pd.date_range('2012-08-02', periods=4))
```  
```python
>>> df
   A
2012-08-02  1
2012-08-03  2
2012-08-04  3
2012-08-05  4
```  
To get the difference between each 2-day period’s maximum and minimum value in one pass, you can do

```python
>>> df.resample('2D').pipe(lambda x: x.max() - x.min())
   A
2012-08-02  1
2012-08-04  1
```  

## 3.11.3 Upsampling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Resampler.ffill</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>Resampler.backfill</code></td>
<td>Backward fill the new missing values in the resampled data.</td>
</tr>
<tr>
<td><code>Resampler.bfill</code></td>
<td>Backward fill the new missing values in the resampled data.</td>
</tr>
<tr>
<td><code>Resampler.pad</code></td>
<td>Forward fill the values.</td>
</tr>
<tr>
<td><code>Resampler.nearest</code></td>
<td>Resample by using the nearest value.</td>
</tr>
<tr>
<td><code>Resampler.fillna</code></td>
<td>Fill missing values introduced by upsampling.</td>
</tr>
<tr>
<td><code>Resampler.asfreq</code></td>
<td>Return the values at the new freq, essentially a reindex.</td>
</tr>
<tr>
<td><code>Resampler.interpolate</code></td>
<td>Interpolate values according to different methods.</td>
</tr>
</tbody>
</table>

### pandas.core.resample.Resampler.ffill

Resampler.ffill([limit])

Forward fill the values.

Parameters

- **limit** *(int, optional)* Limit of how many values to fill.

Returns

- An upsampled Series.

See also:

- **Series.fillna** Fill NA/NaN values using the specified method.
- **DataFrame.fillna** Fill NA/NaN values using the specified method.
pandas.core.resample.Resampler.backfill

Resampler.backfill(limit=None)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

Parameters

- limit [int, optional] Limit of how many values to fill.

Returns

- Series, DataFrame An upsampled Series or DataFrame with backward filled NaN values.

See also:

- bfill Alias of backfill.
- fillna Fill NaN values using the specified method, which can be ‘backfill’.
- nearest Fill NaN values with nearest neighbor starting from center.
- ffill Forward fill NaN values.
- Series.fillna Fill NaN values in the Series using the specified method, which can be ‘backfill’.
- DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be ‘backfill’.

References

[1]

Examples

Resampling a Series:

```python
def main():
    s = pd.Series([1, 2, 3],
                   index=pd.date_range('20180101', periods=3, freq='h'))

    print(s)
    print(s.resample('30min').bfill())
    print(s.resample('15min').bfill(limit=2))

if __name__ == '__main__':
    main()
```

(continues on next page)
Resampling a DataFrame that has missing values:

```python
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
                    index=pd.date_range('20180101', periods=3, freq='h'))
>>> df
     a  b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 02:00:00  6.0  5.0

>>> df.resample('30min').bfill()
     a  b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:30:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:30:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0

>>> df.resample('15min').bfill(limit=2)
     a  b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  NaN
2018-01-01 01:00:00  NaN  NaN
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
```

**pandas.core.resample.Resampler.bfill**

Resampler.bfill(limit=None)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

**Parameters**

- **limit** [int, optional] Limit of how many values to fill.

**Returns**
Series, DataFrame  An upsampled Series or DataFrame with backward filled NaN values.

See also:
bfill  Alias of backfill.
fillna  Fill NaN values using the specified method, which can be ‘backfill’.
nearest  Fill NaN values with nearest neighbor starting from center.
ffill  Forward fill NaN values.
Series.fillna  Fill NaN values in the Series using the specified method, which can be ‘backfill’.
DataFrame.fillna  Fill NaN values in the DataFrame using the specified method, which can be ‘backfill’.

References
[1]

Examples

Resampling a Series:

```python
>>> s = pd.Series([1, 2, 3],
                 index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00 1
2018-01-01 01:00:00 2
2018-01-01 02:00:00 3
Freq: H, dtype: int64
```

```python
>>> s.resample('30min').bfill()
2018-01-01 00:00:00 1
2018-01-01 00:30:00 2
2018-01-01 01:00:00 2
2018-01-01 01:30:00 3
2018-01-01 02:00:00 3
Freq: 30T, dtype: int64
```

```python
>>> s.resample('15min').bfill(limit=2)
2018-01-01 00:00:00 1.0
2018-01-01 00:15:00 NaN
2018-01-01 00:30:00 2.0
2018-01-01 00:45:00 2.0
2018-01-01 01:00:00 2.0
2018-01-01 01:15:00 NaN
2018-01-01 01:30:00 3.0
2018-01-01 01:45:00 3.0
2018-01-01 02:00:00 3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```python
>>> df = pd.DataFrame({
'a': [2, np.nan, 6],
'b': [1, 3, 5],
},
index=pd.date_range('20180101', periods=3, freq='h'))
```

(continues on next page)
a b
2018-01-01 00:00:00 2.0 1
2018-01-01 01:00:00 NaN 3
2018-01-01 02:00:00 6.0 5

```python
df.resample('30min').bfill()
da b
2018-01-01 00:00:00 2.0 1
2018-01-01 00:30:00 NaN 3
2018-01-01 01:00:00 NaN 3
2018-01-01 01:30:00 6.0 5
2018-01-01 02:00:00 6.0 5
```

```python
df.resample('15min').bfill(limit=2)
da b
2018-01-01 00:00:00 2.0 1.0
2018-01-01 00:15:00 NaN NaN
2018-01-01 00:30:00 NaN 3.0
2018-01-01 00:45:00 NaN 3.0
2018-01-01 01:00:00 NaN 3.0
2018-01-01 01:15:00 NaN NaN
2018-01-01 01:30:00 6.0 5.0
2018-01-01 01:45:00 6.0 5.0
2018-01-01 02:00:00 6.0 5.0
```

**pandas.core.resample.Resampler.pad**

Resampler.pad(limit=None)

Forward fill the values.

**Parameters**

- **limit** [int, optional] Limit of how many values to fill.

**Returns**

An upsampled Series.

**See also:**

- Series.fillna Fill NA/NaN values using the specified method.
- DataFrame.fillna Fill NA/NaN values using the specified method.

**pandas.core.resample.Resampler.nearest**

Resampler.nearest(limit=None)

Resample by using the nearest value.

When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The nearest method will replace NaN values that appeared in the resampled data with the value from the nearest member of the sequence, based on the index value. Missing values that existed in the original data will not be modified. If limit is given, fill only this many values in each direction for each of the original values.

**Parameters**
Limit of how many values to fill.

Returns

Series or DataFrame An upsampled Series or DataFrame with NaN values filled with their nearest value.

See also:

backfill Backward fill the new missing values in the resampled data.
pad Forward fill NaN values.

Examples

```python
>>> s = pd.Series([1, 2],
                 index=pd.date_range('20180101', periods=2, freq='1h'))
>>> s
2018-01-01 00:00:00 1
2018-01-01 01:00:00 2
Freq: H, dtype: int64

>>> s.resample('15min').nearest()
2018-01-01 00:00:00  1
2018-01-01 00:15:00  1
2018-01-01 00:30:00  2
2018-01-01 00:45:00  2
2018-01-01 01:00:00  2
Freq: 15T, dtype: int64

Limit the number of upsampled values imputed by the nearest:

```python
>>> s.resample('15min').nearest(limit=1)
2018-01-01 00:00:00  1.0
2018-01-01 00:15:00  1.0
2018-01-01 00:30:00  NaN
2018-01-01 00:45:00  2.0
2018-01-01 01:00:00  2.0
Freq: 15T, dtype: float64
```

**pandas.core.resample.Resampler.fillna**

`Resampler.fillna(method, limit=None)`

Fill missing values introduced by up sampling.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency).

Missing values that existed in the original data will not be modified.

Parameters

  - ‘pad’ or ‘ffill’: use previous valid observation to fill gap (forward fill).
• ‘backfill’ or ‘bfill’: use next valid observation to fill gap.
• ‘nearest’: use nearest valid observation to fill gap.

**limit** [int, optional] Limit of how many consecutive missing values to fill.

**Returns**

Series or DataFrame An upsampled Series or DataFrame with missing values filled.

**See also:**

bfill Backward fill NaN values in the resampled data.
ffill Forward fill NaN values in the resampled data.
nearest Fill NaN values in the resampled data with nearest neighbor starting from center.
interpolate Fill NaN values using interpolation.
Series.fillna Fill NaN values in the Series using the specified method, which can be ‘bfill’ and ‘ffill’.
DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be ‘bfill’ and ‘ffill’.

**References**

[1]

**Examples**

Resampling a Series:

```python
>>> s = pd.Series([1, 2, 3],
                 index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

Without filling the missing values you get:

```python
>>> s.resample("30min").asfreq()
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    2.0
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```python
>>> s.resample('30min').fillna('backfill')
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```
>>> s.resample('15min').fillna("backfill", limit=2)
2018-01-01 00:00:00  1.0
2018-01-01 00:15:00  NaN
2018-01-01 00:30:00  2.0
2018-01-01 00:45:00  2.0
2018-01-01 01:00:00  2.0
2018-01-01 01:15:00  NaN
2018-01-01 01:30:00  3.0
2018-01-01 01:45:00  3.0
2018-01-01 02:00:00  3.0
Freq: 15T, dtype: float64

>>> s.resample('30min').fillna("pad")
2018-01-01 00:00:00  1
2018-01-01 00:30:00  1
2018-01-01 01:00:00  2
2018-01-01 01:30:00  2
2018-01-01 02:00:00  3
Freq: 30T, dtype: int64

>>> s.resample('30min').fillna("nearest")
2018-01-01 00:00:00  1
2018-01-01 00:30:00  2
2018-01-01 01:00:00  2
2018-01-01 01:30:00  3
2018-01-01 02:00:00  3
Freq: 30T, dtype: int64

Missing values present before the upsampling are not affected.

>>> sm = pd.Series([1, None, 3], ... index=pd.date_range('20180101', periods=3, freq='h'))

>>> sm
2018-01-01 00:00:00  1.0
2018-01-01 01:00:00  NaN
2018-01-01 02:00:00  3.0
Freq: H, dtype: float64

>>> sm.resample('30min').fillna('backfill')
2018-01-01 00:00:00  1.0
2018-01-01 00:30:00  NaN
2018-01-01 01:00:00  NaN
2018-01-01 01:30:00  3.0
2018-01-01 02:00:00  3.0
Freq: 30T, dtype: float64

>>> sm.resample('30min').fillna('pad')
2018-01-01 00:00:00  1.0
2018-01-01 00:30:00  1.0
2018-01-01 01:00:00  NaN
2018-01-01 01:30:00  NaN
2018-01-01 02:00:00  3.0
Freq: 30T, dtype: float64

```python
>>> sm.resample('30min').fillna('nearest')
2018-01-01 00:00:00 1.0
2018-01-01 00:30:00 NaN
2018-01-01 01:00:00 NaN
2018-01-01 01:30:00 3.0
2018-01-01 02:00:00 3.0
Freq: 30T, dtype: float64
```

DataFrame resampling is done column-wise. All the same options are available.

```python
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
                   index=pd.date_range('20180101', periods=3,
                   freq='h'))
>>> df
   a   b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 01:00:00 NaN  3.0
2018-01-01 02:00:00  6.0  5.0
```

```python
>>> df.resample('30min').fillna('bfill')
   a    b
2018-01-01 00:00:00 2.0  1.0
2018-01-01 00:30:00 NaN  3.0
2018-01-01 01:00:00 NaN  3.0
2018-01-01 01:30:00 6.0  5.0
2018-01-01 02:00:00 6.0  5.0
```

```python
pandas.core.resample.Resampler.asfreq
```

Resampler.asfreq(fill_value=None)

Return the values at the new freq, essentially a reindex.

Parameters

fill_value [scalar, optional] Value to use for missing values, applied during upampling (note
this does not fill NaNs that already were present).

Returns

DataFrame or Series Values at the specified freq.

See also:

Series.asfreq Convert TimeSeries to specified frequency.
DataFrame.asfreq Convert TimeSeries to specified frequency.
pandas.core.resample.Resampler.interpolate

Resampler.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', limit_area=None, downcast=None, **kwargs)

Interpolate values according to different methods.

Fill NaN values using an interpolation method.

Please note that only method='linear' is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default ‘linear’] Interpolation technique to use. One of:

• ‘linear’: Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
• ‘time’: Works on daily and higher resolution data to interpolate given length of interval.
• ‘index’, ‘values’: use the actual numerical values of the index.
• ‘pad’: Fill in NaNs using existing values.
• ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘spline’, ‘barycentric’, ‘polynomial’: Passed to scipy.interpolate.interp1d. These methods use the numerical values of the index. Both ‘polynomial’ and ‘spline’ require that you also specify an order (int), e.g. df.interpolate(method='polynomial', order=5).
• ‘from_derivatives’: Refers to scipy.interpolate.BPoly.from_derivatives which replaces ‘piecewise_polynomial’ interpolation method in scipy 0.18.

axis [{0 or ‘index’, 1 or ‘columns’, None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{‘forward’, ‘backward’, ‘both’}, Optional] Consecutive NaNs will be filled in this direction.

If limit is specified:

• If ‘method’ is ‘pad’ or ‘ffill’, ‘limit_direction’ must be ‘forward’.
• If ‘method’ is ‘backfill’ or ‘bfill’, ‘limit_direction’ must be ‘backwards’.

If ‘limit’ is not specified:

• If ‘method’ is ‘backfill’ or ‘bfill’, the default is ‘backward’
• else the default is ‘forward’

Changed in version 1.1.0: raises ValueError if limit_direction is ‘forward’ or ‘both’ and method is ‘backfill’ or ‘bfill’. raises ValueError if limit_direction is ‘backward’ or ‘both’ and method is ‘pad’ or ‘ffill’.

limit_area [{None, ‘inside’, ‘outside’}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

• None: No fill restriction.
• ‘inside’: Only fill NaNs surrounded by valid values (interpolate).
• ‘outside’: Only fill NaNs outside valid values (extrapolate).

downcast  [optional, ‘infer’ or None, defaults to None] Downcast dtypes if possible.

````kwargs```` [optional] Keyword arguments to pass on to the interpolating function.

Returns

**Series or DataFrame or None**  Returns the same object type as the caller, interpolated at some or all NaN values or None if inplace=True.

See also:

fillna  Fill missing values using different methods.
scipy.interpolate.Akima1DInterpolator  Piecewise cubic polynomials (Akima interpolator).
scipy.interpolate.BPoly.from_derivatives  Piecewise polynomial in the Bernstein basis.
scipy.interpolate.interp1d  Interpolate a 1-D function.
scipy.interpolate.KroghInterpolator  Interpolate polynomial (Krogh interpolator).
scipy.interpolate.PchipInterpolator  PCHIP 1-d monotonic cubic interpolation.
scipy.interpolate.CubicSpline  Cubic spline data interpolator.

Notes

The ‘krogh’, ‘piecewise_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in NaN in a Series via linear interpolation.

```python
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0 0.0
1 1.0
2 NaN
3 3.0
dtype: float64
```
```
>>> s.interpolate()
0 0.0
1 1.0
2 2.0
3 3.0
dtype: float64
```

Filling in NaN in a Series by padding, but filling at most two consecutive NaN at a time.

```python
>>> s = pd.Series([np.nan, "single_one", np.nan,
... "fill_two_more", np.nan, np.nan, np.nan,
... 4.71, np.nan])
```
```python
>>> s
0      NaN
1    single_one
2      NaN
3  fill_two_more
4      NaN
```
(continues on next page)
Filling in NaN in a Series via polynomial interpolation or splines: Both ‘polynomial’ and ‘spline’ methods require that you also specify an order (int).

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column ‘a’ is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column ‘b’ remains NaN, because there is no entry before it to use for interpolation.

Using polynomial interpolation.
```python
>>> df['d'].interpolate(method='polynomial', order=2)
0    1.0
1    4.0
2    9.0
3   16.0
Name: d, dtype: float64
```

### 3.11.4 Computations / descriptive stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Resampler.count()</code></td>
<td>Compute count of group, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.nunique()</code></td>
<td>Return number of unique elements in the group.</td>
</tr>
<tr>
<td><code>Resampler.first()</code></td>
<td>Compute first of group values.</td>
</tr>
<tr>
<td><code>Resampler.last()</code></td>
<td>Compute last of group values.</td>
</tr>
<tr>
<td><code>Resampler.max()</code></td>
<td>Compute max of group values.</td>
</tr>
<tr>
<td><code>Resampler.mean()</code></td>
<td>Compute mean of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.median()</code></td>
<td>Compute median of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.min()</code></td>
<td>Compute min of group values.</td>
</tr>
<tr>
<td><code>Resampler.ohlc()</code></td>
<td>Compute open, high, low and close values of a group, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.prod()</code></td>
<td>Compute prod of group values.</td>
</tr>
<tr>
<td><code>Resampler.size()</code></td>
<td>Compute group sizes.</td>
</tr>
<tr>
<td><code>Resampler.sem()</code></td>
<td>Compute standard error of the mean of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.std()</code></td>
<td>Compute standard deviation of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.sum()</code></td>
<td>Compute sum of group values.</td>
</tr>
<tr>
<td><code>Resampler.var()</code></td>
<td>Compute variance of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>Resampler.quantile()</code></td>
<td>Return value at the given quantile.</td>
</tr>
</tbody>
</table>

**pandas.core.resample.Resampler.count**

Resampler.count()

Compute count of group, excluding missing values.

**Returns**

- `Series` or `DataFrame`: Count of values within each group.

**See also:**

- `Series.groupby`: Apply a function groupby to a Series.
- `DataFrame.groupby`: Apply a function groupby to each row or column of a DataFrame.
pandas.core.resample.Resampler.nunique

Resampler.nunique(_method='nunique')

Return number of unique elements in the group.

Returns

Series  Number of unique values within each group.

pandas.core.resample.Resampler.first

Resampler.first(_method='first', min_count=0, *args, **kwargs)

Compute first of group values.

Parameters

numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

Returns

Series or DataFrame  Computed first of values within each group.

pandas.core.resample.Resampler.last

Resampler.last(_method='last', min_count=0, *args, **kwargs)

Compute last of group values.

Parameters

numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

Returns

Series or DataFrame  Computed last of values within each group.

pandas.core.resample.Resampler.max

Resampler.max(_method='max', min_count=0, *args, **kwargs)

Compute max of group values.

Parameters

numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count [int, default -1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

Returns

Series or DataFrame  Computed max of values within each group.
pandas.core.resample.Resampler.mean

Resampler.mean(_method=’mean’, *args, **kwargs)

Compute mean of groups, excluding missing values.

Parameters

- **numeric_only** [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.
- **engine** [str, default None]
  - 'cython': Runs the operation through C-extensions from cython.
  - 'numba': Runs the operation through JIT compiled code from numba.
  - None: Defaults to 'cython' or globally setting compute.use_numba

New in version 1.4.0.

- **engine_kwargs** [dict, default None]
  - For 'cython' engine, there are no accepted engine_kwargs
  - For 'numba' engine, the engine can accept nopython, nogil and parallel dictionary keys. The values must either be True or False. The default engine_kwargs for the 'numba' engine is {{'nopython': True, 'nogil': False, 'parallel': False}}

New in version 1.4.0.

Returns

pandas.Series or pandas.DataFrame

See also:

- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                     'B': [np.nan, 2, 3, 4, 5],
...                     'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```python
>>> df.groupby('A').mean()
   B     C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```python
>>> df.groupby(['A', 'B']).mean()
      C
A B
1 2.0  2.0
4.0  1.0
```

(continues on next page)
Groupby one column and return the mean of only particular column in the group.

```python
>>> df.groupby('A')['B'].mean()
A
1  3.0
2  4.0
Name: B, dtype: float64
```

**pandas.core.resample.Resampler.median**

Resampler.median(_method='median', *args, **kwargs)

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

**Parameters**

- numeric_only [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

**Returns**

- Series or DataFrame Median of values within each group.

**See also:**

- Series.groupby Apply a function groupby to a Series.
- DataFrame.groupby Apply a function groupby to each row or column of a DataFrame.

**pandas.core.resample.Resampler.min**

Resampler.min(_method='min', min_count=0, *args, **kwargs)

Compute min of group values.

**Parameters**

- numeric_only [bool, default False] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

- min_count [int, default 1] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

**Returns**

- Series or DataFrame Computed min of values within each group.
pandas.core.resample.Resampler.ohlc

Resampler.ohlc(_method='ohlc', *args, **kwargs)

Compute open, high, low and close values of a group, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Returns

DataFrame  Open, high, low and close values within each group.

See also:

Series.groupby  Apply a function groupby to a Series.
DataFrame.groupby  Apply a function groupby to each row or column of a DataFrame.

pandas.core.resample.Resampler.prod

Resampler.prod(_method='prod', min_count=0, *args, **kwargs)

Compute prod of group values.

Parameters

numeric_only  [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count  [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

Returns

Series or DataFrame  Computed prod of values within each group.

pandas.core.resample.Resampler.size

Resampler.size()

Compute group sizes.

Returns

DataFrame or Series  Number of rows in each group as a Series if as_index is True or a DataFrame if as_index is False.

See also:

Series.groupby  Apply a function groupby to a Series.
DataFrame.groupby  Apply a function groupby to each row or column of a DataFrame.

pandas.core.resample.Resampler.sem

Resampler.sem(_method='sem', *args, **kwargs)

Compute standard error of the mean of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof  [int, default 1] Degrees of freedom.

Returns

Series or DataFrame  Standard error of the mean of values within each group.

See also:
Series.groupby
Apply a function groupby to a Series.

DataFrame.groupby
Apply a function groupby to each row or column of a DataFrame.

pandas.core.resample.Resampler.std

Resampler.std(ddof=1, *args, **kwargs)
Compute standard deviation of groups, excluding missing values.

Parameters

ddof [int, default 1] Degrees of freedom.

Returns

DataFrame or Series Standard deviation of values within each group.

pandas.core.resample.Resampler.sum

Resampler.sum(_method='sum', min_count=0, *args, **kwargs)
Compute sum of group values.

Parameters

numeric_only [bool, default True] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA.

Returns

Series or DataFrame Computed sum of values within each group.

pandas.core.resample.Resampler.var

Resampler.var(ddof=1, *args, **kwargs)
Compute variance of groups, excluding missing values.

Parameters

ddof [int, default 1] Degrees of freedom.

Returns

DataFrame or Series Variance of values within each group.

pandas.core.resample.Resampler.quantile

Resampler.quantile(q=0.5, **kwargs)
Return value at the given quantile.

Parameters

q [float or array-like, default 0.5 (50% quantile)]

Returns

DataFrame or Series Quantile of values within each group.

See also:

Series.quantile Return a series, where the index is q and the values are the quantiles.
DataFrame.quantile  Return a DataFrame, where the columns are the columns of self, and the values are the quantiles.

DataFrameGroupBy.quantile  Return a DataFrame, where the columns are groupby columns, and the values are its quantiles.

3.12 Style

Styler objects are returned by pandas.DataFrame.style.

3.12.1 Styler constructor

```
Styler(data[, precision, table_styles, ...])  Helps style a DataFrame or Series according to the data with HTML and CSS.
Styler.from_custom_template(searchpath[, ...])  Factory function for creating a subclass of Styler.
```

pandas.io.formats.style.Styler

class pandas.io.formats.style.Styler(data, precision=None, table_styles=None, uuid=None, caption=None, table_attributes=None, cell_ids=True, na_rep=None, uuid_len=5, decimal=None, thousands=None, escape=None, formatter=None)

Helps style a DataFrame or Series according to the data with HTML and CSS.

Parameters

data  [Series or DataFrame] Data to be styled - either a Series or DataFrame.

precision  [int, optional] Precision to round floats to. If not given defaults to pandas.options.styler.format.precision. Changed in version 1.4.0.

table_styles  [list-like, default None] List of \{selector: (attr, value)\} dicts; see Notes.

uuid  [str, default None] A unique identifier to avoid CSS collisions; generated automatically.

caption  [str, tuple, default None] String caption to attach to the table. Tuple only used for LaTeX dual captions.

table_attributes  [str, default None] Items that show up in the opening <table> tag in addition to automatic (by default) id.

cell_ids  [bool, default True] If True, each cell will have an id attribute in their HTML tag. The id takes the form T_<uuid>_row<num_row>_col<num_col> where <uuid> is the unique identifier, <num_row> is the row number and <num_col> is the column number.

na_rep  [str, optional] Representation for missing values. If na_rep is None, no special formatting is applied, and falls back to pandas.options.styler.format.na_rep. New in version 1.0.0.

uuid_len  [int, default 5] If uuid is not specified, the length of the uuid to randomly generate expressed in hex characters, in range [0, 32]. New in version 1.2.0.
decimal [str, optional] Character used as decimal separator for floats, complex and integers. If not given uses pandas.optionsstyler.format.decimal.

New in version 1.3.0.

thousands [str, optional, default None] Character used as thousands separator for floats, complex and integers. If not given uses pandas.optionsstyler.format.thousands.

New in version 1.3.0.

escape [str, optional] Use ‘html’ to replace the characters & , <, , ', and " in cell display string with HTML-safe sequences. Use ‘latex’ to replace the characters &<, $, #, _{}, ^, and \ in the cell display string with LaTeX-safe sequences. If not given uses pandas.optionsstyler.format.escape.

New in version 1.3.0.

formatter [str, callable, dict, optional] Object to define how values are displayed. See Styler.format. If not given uses pandas.optionsstyler.format.formatter.

New in version 1.4.0.

See also:

DataFrame.style Return a Styler object containing methods for building a styled HTML representation for the DataFrame.

Notes

Most styling will be done by passing style functions into Styler.apply or Styler.applymap. Style functions should return values with strings containing CSS ‘attr: value’ that will be applied to the indicated cells.

If using in the Jupyter notebook, Styler has defined a _repr_html_ to automatically render itself. Otherwise call Styler.to_html to get the generated HTML.

CSS classes are attached to the generated HTML

- Index and Column names include index_name and level<k> where k is its level in a MultiIndex
- Index label cells include
  - row_heading
  - row<n> where n is the numeric position of the row
  - level<k> where k is the level in a MultiIndex
- Column label cells include *col_heading* *col<n> where n is the numeric position of the column *level<k> where k is the level in a MultiIndex
- Blank cells include blank
- Data cells include data
- Trimmed cells include col_trim or row_trim.

Any, or all, or these classes can be renamed by using the css_class_names argument in Styler. set_table_classes, giving a value such as {“row”: “MY_ROW_CLASS”, “col_trim”: “”, “row_trim”: “”}. 3.12. Style 2635
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>(Jinja2.jinja2.Environment)</td>
</tr>
<tr>
<td>template_html</td>
<td>(Jinja2.Template)</td>
</tr>
<tr>
<td>template_html_table</td>
<td>(Jinja2.Template)</td>
</tr>
<tr>
<td>template_html_style</td>
<td>(Jinja2.Template)</td>
</tr>
<tr>
<td>template_latex</td>
<td>(Jinja2.Template)</td>
</tr>
<tr>
<td>loader</td>
<td>(Jinja2.Loader)</td>
</tr>
</tbody>
</table>

Methods

- `apply(func[, axis, subset])`  
  Apply a CSS-styling function column-wise, row-wise, or table-wise.

- `apply_index(func[, axis, level])`  
  Apply a CSS-styling function to the index or column headers, level-wise.

- `applymap(func[, subset])`  
  Apply a CSS-styling function elementwise.

- `applymap_index(func[, axis, level])`  
  Apply a CSS-styling function to the index or column headers, elementwise.

- `background_gradient([cmap, low, high, axis,...])`  
  Color the background in a gradient style.

- `bar([subset, axis, color, cmap, width,...])`  
  Draw bar chart in the cell backgrounds.

- `clear()`  
  Reset the Styler, removing any previously applied styles.

- `export()`  
  Export the styles applied to the current Styler.

- `format([formatter, subset, na_rep,...])`  
  Format the text display value of cells.

- `format_index([formatter, axis, level,...])`  
  Format the text display value of index labels or column headers.

- `from_custom_template(searchpath[, ...])`  
  Factory function for creating a subclass of Styler.

- `hide([subset, axis, level, names])`  
  Hide the entire index / column headers, or specific rows / columns from display.

- `hide_columns([subset, level, names])`  
  Hide the column headers or specific keys in the columns from rendering.

- `hide_index([subset, level, names])`  
  (DEPRECATED) Hide the entire index, or specific keys in the index from rendering.

- `highlight_between([subset, color, axis,...])`  
  Highlight a defined range with a style.

- `highlight_max([subset, color, axis, props])`  
  Highlight the maximum with a style.

- `highlight_min([subset, color, axis, props])`  
  Highlight the minimum with a style.

- `highlight_null([null_color, subset, props])`  
  Highlight missing values with a style.

- `highlight_quantile([subset, color, axis,...])`  
  Highlight values defined by a quantile with a style.

- `pipe(func, *args, **kwargs)`  
  Apply `func(self, *args, **kwargs)`, and return the result.

- `render([sparse_index, sparse_columns])`  
  (DEPRECATED) Render the Styler including all applied styles to HTML.

- `set_caption(caption)`  
  Set the text added to a `<caption>` HTML element.

- `set_na_rep(na_rep)`  
  (DEPRECATED) Set the missing data representation on a Styler.

- `set_precision(precision)`  
  (DEPRECATED) Set the precision used to display values.

- `set_properties([subset])`  
  Set defined CSS-properties to each `<td>` HTML element within the given subset.

continues on next page
Table 13 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_sticky</code>([axis, pixel_size, levels])</td>
<td>Add CSS to permanently display the index or column headers in a scrolling frame.</td>
</tr>
<tr>
<td><code>set_table_attributes</code>(attributes)</td>
<td>Set the table attributes added to the <code>&lt;table&gt;</code> HTML element.</td>
</tr>
<tr>
<td><code>set_table_styles</code>((table_styles, axis, ...))</td>
<td>Set the table styles included within the <code>&lt;style&gt;</code> HTML element.</td>
</tr>
<tr>
<td><code>set_td_classes</code>(classes)</td>
<td>Set the DataFrame of strings added to the class attribute of <code>&lt;td&gt;</code> HTML elements.</td>
</tr>
<tr>
<td><code>set_tooltips</code>(ttips[, props, css_class])</td>
<td>Set the DataFrame of strings on Styler generating :hover tooltips.</td>
</tr>
<tr>
<td><code>set_uuid</code>(uuid)</td>
<td>Set the uuid applied to <code>id</code> attributes of HTML elements.</td>
</tr>
<tr>
<td><code>text_gradient</code>([cmap, low, high, axis, ...])</td>
<td>Color the text in a gradient style.</td>
</tr>
<tr>
<td><code>to_excel</code>(excel_writer[, sheet_name, na_rep, ...])</td>
<td>Write Styler to an Excel sheet.</td>
</tr>
<tr>
<td><code>to_html</code>([buf, table_uuid, table_attributes, ...])</td>
<td>Write Styler to a file, buffer or string in HTML-CSS format.</td>
</tr>
<tr>
<td><code>to_latex</code>([buf, column_format, position, ...])</td>
<td>Write Styler to a file, buffer or string in LaTeX format.</td>
</tr>
<tr>
<td><code>use</code>(styles)</td>
<td>Set the styles on the current Styler.</td>
</tr>
<tr>
<td><code>where</code>(cond, value[, other, subset])</td>
<td>(DEPRECATED) Apply CSS-styles based on a conditional function elementwise.</td>
</tr>
</tbody>
</table>

**pandas.io.formats.style.Styler.apply**

Styler.apply(`func`, `axis`=0, `subset`=None, **kwargs)

Apply a CSS-styling function column-wise, row-wise, or table-wise.

Updates the HTML representation with the result.

Parameters

- `func` [function] `func` should take a Series if `axis` in [0,1] and return a list-like object of same length, or a Series, not necessarily of same length, with valid index labels considering `subset`. `func` should take a DataFrame if `axis` is `None` and return either an ndarray with the same shape or a DataFrame, not necessarily of the same shape, with valid index and columns labels considering `subset`.

  Changed in version 1.3.0.

  Changed in version 1.4.0.

- `axis` [(0 or 'index', 1 or 'columns', `None`), default 0] Apply to each column (`axis`=0 or 'index'), to each row (`axis`=1 or 'columns'), or to the entire DataFrame at once with `axis`=None.

- `subset` [label, array-like, IndexSlice, optional] A valid 2d input to `DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to `DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to before applying the function.

  **kwargs [dict] Pass along to `func`.

Returns

- `self` [Styler]

See also:
Styler.applymap_index  Apply a CSS-styling function to headers elementwise.
Styler.apply_index  Apply a CSS-styling function to headers level-wise.
Styler.applymap  Apply a CSS-styling function elementwise.

Notes

The elements of the output of func should be CSS styles as strings, in the format 'attribute: value; attribute2: value2; ...' or, if nothing is to be applied to that element, an empty string or None.

This is similar to DataFrame.apply, except that axis=None applies the function to the entire DataFrame at once, rather than column-wise or row-wise.

Examples

```python
>>> def highlight_max(x, color):
...     return np.where(x == np.nanmax(x.to_numpy()), f"color: {color};", None)
>>> df = pd.DataFrame(np.random.randn(5, 2), columns=["A", "B"])
>>> df.style.apply(highlight_max, color='red')
>>> df.style.apply(highlight_max, color='blue', axis=1)
>>> df.style.apply(highlight_max, color='green', axis=None)
```

Using subset to restrict application to a single column or multiple columns

```python
>>> df.style.apply(highlight_max, color='red', subset="A")
... >>> df.style.apply(highlight_max, color='red', subset=["A", "B"])
... ...
```

Using a 2d input to subset to select rows in addition to columns

```python
>>> df.style.apply(highlight_max, color='red', subset=(0,1,2), slice(None))
... >>> df.style.apply(highlight_max, color='red', subset=(slice(0,5,2), "A"))
... ...
```

Using a function which returns a Series / DataFrame of unequal length but containing valid index labels

```python
>>> total_style = pd.Series("font-weight: bold;", index=["Total"])
>>> df.style.apply(lambda s: total_style)
```

See Table Visualization user guide for more details.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.io.formats.style.Styler.apply_index

Styler.apply_index(func, axis=0, level=None, **kwargs)

Apply a CSS-styling function to the index or column headers, level-wise.

Updates the HTML representation with the result.

New in version 1.4.0.

Parameters

- **func** [function] func should take a Series and return a string array of the same length.
- **axis** [{0, 1, “index”, “columns”}] The headers over which to apply the function.
- **level** [int, str, list, optional] If index is MultiIndex the level(s) over which to apply the function.
- **kwargs** [dict] Pass along to func.

Returns

self [Styler]

See also:

Styler.applymap_index Apply a CSS-styling function to headers elementwise.
Styler.apply Apply a CSS-styling function column-wise, row-wise, or table-wise.
Styler.applymap Apply a CSS-styling function elementwise.

Notes

Each input to func will be the index as a Series, if an Index, or a level of a MultiIndex. The output of func should be an identically sized array of CSS styles as strings, in the format ‘attribute: value; attribute2: value2; …’ or, if nothing is to be applied to that element, an empty string or None.

Examples

Basic usage to conditionally highlight values in the index.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], index=\"A", \"B\")
>>> def color_b(s):
...     return np.where(s == \"B\", \"background-color: yellow;\", \"")
>>> df.style.apply_index(color_b)
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Selectively applying to specific levels of MultiIndex columns.

3.12. Style
```python
>>> midx = pd.MultiIndex.from_product([['ix', 'jy'], [0, 1], ['x3', 'z4']])
>>> df = pd.DataFrame([np.arange(8)], columns=midx)
>>> def highlight_x(s):
...     return "background-color: yellow;" if "x" in v else "" for v in s
>>> df.style.apply_index(highlight_x, axis="columns", level=[0, 2])
```

<table>
<thead>
<tr>
<th></th>
<th>ix</th>
<th>jy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x3</td>
<td>z4</td>
<td>x3</td>
</tr>
<tr>
<td>x3</td>
<td>z4</td>
<td>x4</td>
</tr>
</tbody>
</table>

```python
pandas.io.formats.style.Styler.applymap

Styler.applymap(func, subset=None, **kwargs)

Apply a CSS-styling function elementwise.

Updates the HTML representation with the result.

**Parameters**

- **func** [function] func should take a scalar and return a string.
- **subset** [label, array-like, IndexSlice, optional] A valid 2d input to `DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to `DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to before applying the function.
- **kwargs** [dict] Pass along to func.

**Returns**

- **self** [Styler]

See also:

- **Styler.applymap_index** Apply a CSS-styling function to headers elementwise.
- **Styler.apply_index** Apply a CSS-styling function to headers level-wise.
- **Styler.apply** Apply a CSS-styling function column-wise, row-wise, or table-wise.

**Notes**

The elements of the output of func should be CSS styles as strings, in the format ‘attribute: value; attribute2: value2; …’ or, if nothing is to be applied to that element, an empty string or None.
Examples

```python
>>> def color_negative(v, color):
...     return f"color: {color};" if v < 0 else None

>>> df = pd.DataFrame(np.random.randn(5, 2), columns="A", "B")

>>> df.style.applymap(color_negative, color='red')

Using subset to restrict application to a single column or multiple columns

```python

```python
>>> df.style.applymap(color_negative, color='red', subset="A")

>>> df.style.applymap(color_negative, color='red', subset=["A", "B"])

Using a 2d input to subset to select rows in addition to columns

```python

```python
>>> df.style.applymap(color_negative, color='red',
... subset=[[0,1,2], slice(None)])

>>> df.style.applymap(color_negative, color='red', subset=(slice(0,5,2), "A"))

See `Table Visualization` user guide for more details.

**pandas.io.formats.style.Styler.applymap_index**

**Styler.applymap_index**(func, axis=0, level=None, **kwargs)

Apply a CSS-styling function to the index or column headers, elementwise.

Updates the HTML representation with the result.

New in version 1.4.0.

**Parameters**

- **func** [function] func should take a scalar and return a string.
- **axis** [[0, 1, “index”, “columns”]] The headers over which to apply the function.
- **level** [int, str, list, optional] If index is MultiIndex the level(s) over which to apply the function.
- ****kwargs** [dict] Pass along to func.

**Returns**

- **self** [Styler]

See also:

- **Styler.apply_index** Apply a CSS-styling function to headers level-wise.
- **Styler.apply** Apply a CSS-styling function column-wise, row-wise, or table-wise.
- **Styler.applymap** Apply a CSS-styling function elementwise.
Notes

Each input to `func` will be an index value, if an Index, or a level value of a MultiIndex. The output of `func` should be CSS styles as a string, in the format ‘attribute: value; attribute2: value2;…’ or, if nothing is to be applied to that element, an empty string or `None`.

Examples

Basic usage to conditionally highlight values in the index.

```python
>>> df = pd.DataFrame([ [1,2], [3,4] ], index=["A", "B"])
>>> def color_b(s):
...     return "background-color: yellow;" if v == "B" else None
>>> df.style.applymap_index(color_b)
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Selectively applying to specific levels of MultiIndex columns.

```python
>>> midx = pd.MultiIndex.from_product([['ix', 'jy'], [0, 1], ['x3', 'z4']])
>>> df = pd.DataFrame([np.arange(8)], columns=midx)
>>> def highlight_x(v):
...     return "background-color: yellow;" if "x" in v else None
>>> df.style.applymap_index(highlight_x, axis="columns", level=[0, 2])
```

```
<table>
<thead>
<tr>
<th></th>
<th>ix</th>
<th>jy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>x3</td>
<td>z4</td>
</tr>
<tr>
<td></td>
<td>x3</td>
<td>z4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

`pandas.io.formats.style.Styler.background_gradient`

Styler.method_background_gradient(*cmap='PuBu', low=0, high=0, axis=0, subset=None, text_color_threshold=0.408, vmin=None, vmax=None, gmap=None*)

Color the background in a gradient style.

The background color is determined according to the data in each column, row or frame, or by a given gradient map. Requires matplotlib.

Parameters

- `cmap` [str or colormap] Matplotlib colormap.

**low** [float] Compress the color range at the low end. This is a multiple of the data range to extend below the minimum; good values usually in [0, 1], defaults to 0.

**high** [float] Compress the color range at the high end. This is a multiple of the data range to extend above the maximum; good values usually in [0, 1], defaults to 0.

**axis** [[0 or ‘index’, 1 or ‘columns’, None], default 0] Apply to each column (axis=0 or ‘index’), to each row (axis=1 or ‘columns’), or to the entire DataFrame at once with axis=None.

**subset** [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[:, <subset>] where the columns are prioritised, to limit data to before applying the function.

**text_color_threshold** [float or int] Luminance threshold for determining text color in [0, 1]. Facilitates text visibility across varying background colors. All text is dark if 0, and light if 1, defaults to 0.408.

**vmin** [float, optional] Minimum data value that corresponds to colormap minimum value. If not specified the minimum value of the data (or gmap) will be used.

New in version 1.0.0.

**vmax** [float, optional] Maximum data value that corresponds to colormap maximum value. If not specified the maximum value of the data (or gmap) will be used.

New in version 1.0.0.

**gmap** [array-like, optional] Gradient map for determining the background colors. If not supplied will use the underlying data from rows, columns or frame. If given as an ndarray or list-like must be an identical shape to the underlying data considering axis and subset. If given as DataFrame or Series must have same index and column labels considering axis and subset. If supplied, vmin and vmax should be given relative to this gradient map.

New in version 1.3.0.

**Returns**

**self** [Styler]

See also:

**Styler.text_gradient** Color the text in a gradient style.

**Notes**

When using low and high the range of the gradient, given by the data if gmap is not given or by gmap, is extended at the low end effectively by \(\text{map.min} - \text{low} \times \text{map.range}\) and at the high end by \(\text{map.max} + \text{high} \times \text{map.range}\) before the colors are normalized and determined.

If combining with vmin and vmax the \(\text{map.min}, \text{map.max}\) and \(\text{map.range}\) are replaced by values according to the values derived from vmin and vmax.

This method will preselect numeric columns and ignore non-numeric columns unless a gmap is supplied in which case no preselection occurs.
Examples

```python
>>> df = pd.DataFrame(columns=["City", "Temp (c)", "Rain (mm)", "Wind (m/s)"],
                   data=[["Stockholm", 21.6, 5.0, 3.2],
                        ["Oslo", 22.4, 13.3, 3.1],
                        ["Copenhagen", 24.5, 0.0, 6.7]])

Shading the values column-wise, with axis=0, preselecting numeric columns

```python
>>> df.style.background_gradient(axis=0)
```

Shading all values collectively using axis=None

```python
>>> df.style.background_gradient(axis=None)
```

Compress the color map from the both low and high ends

```python
>>> df.style.background_gradient(axis=None, low=0.75, high=1.0)
```

Manually setting vmin and vmax gradient thresholds

```python
>>> df.style.background_gradient(axis=None, vmin=6.7, vmax=21.6)
```

Setting a gmap and applying to all columns with another cmap

```python
>>> df.style.background_gradient(axis=0, gmap=df['Temp (c)'], cmap='YlOrRd')
...```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> gmap = np.array([[1,2,3], [2,3,4], [3,4,5]])
>>> df.style.background_gradient(axis=None, gmap=gmap, ... cmap='YlOrRd', subset=['Temp (c)', 'Rain (mm)', 'Wind (m/s)'] ...)
```

```
City   Temp (c)   Rain (mm)   Wind (m/s)
Stockholm   21.600000  5.000000  3.200000
Oslo       22.400000 13.300000  3.100000
Copenhagen 24.500000  0.000000  6.700000
```

```
City   Temp (c)   Rain (mm)   Wind (m/s)
Stockholm   21.600000  5.000000  3.200000
Oslo       22.400000 13.300000  3.100000
Copenhagen 24.500000  0.000000  6.700000
```

### pandas.io.formats.style.Styler.bar

**Styler.bar**

```
subset=None, axis=0, *, color=None, cmap=None, width=100, height=100, align='mid', vmin=None, vmax=None, props='width: 10em;')
```

Draw bar chart in the cell backgrounds.

Changed in version 1.4.0.

**Parameters**

- **subset** [label, array-like, IndexSlice, optional] A valid 2d input to `DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to `DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to before applying the function.

- **axis** [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (axis=0 or 'index'), to each row (axis=1 or 'columns'), or to the entire DataFrame at once with axis=None.

- **color** [str or 2-tuple/list] If a str is passed, the color is the same for both negative and positive numbers. If 2-tuple/list is used, the first element is the color_negative and the second is the color_positive (eg: ['#d65f5f', '#5fba7d']).

- **cmap** [str, matplotlib.cm.ColorMap] A string name of a matplotlib Colormap, or a Colormap object. Cannot be used together with color.

New in version 1.4.0.
**width** [float, default 100] The percentage of the cell, measured from the left, in which to draw the bars, in [0, 100].

**height** [float, default 100] The percentage height of the bar in the cell, centrally aligned, in [0,100].

New in version 1.4.0.

**align** [str, int, float, callable, default ‘mid’] How to align the bars within the cells relative to a width adjusted center. If string must be one of:

- ‘left’ : bars are drawn rightwards from the minimum data value.
- ‘right’ : bars are drawn leftwards from the maximum data value.
- ‘zero’ : a value of zero is located at the center of the cell.
- ‘mid’ : a value of (max-min)/2 is located at the center of the cell, or if all values are negative (positive) the zero is aligned at the right (left) of the cell.
- ‘mean’ : the mean value of the data is located at the center of the cell.

If a float or integer is given this will indicate the center of the cell.

If a callable should take a 1d or 2d array and return a scalar.

Changed in version 1.4.0.

**vmin** [float, optional] Minimum bar value, defining the left hand limit of the bar drawing range, lower values are clipped to vmin. When None (default): the minimum value of the data will be used.

**vmax** [float, optional] Maximum bar value, defining the right hand limit of the bar drawing range, higher values are clipped to vmax. When None (default): the maximum value of the data will be used.

**props** [str, optional] The base CSS of the cell that is extended to add the bar chart. Defaults to “width: 10em;”.

New in version 1.4.0.

Returns

**self** [Styler]

Notes

This section of the user guide: *Table Visualization* gives a number of examples for different settings and color coordination.

**pandas.io.formats.style.Styler.clear**

*Styler.clear()*

Reset the *Styler*, removing any previously applied styles.

Returns None.
**pandas.io.formats.style.Styler.export**

`Styler.export()`  
Export the styles applied to the current Styler.  
Can be applied to a second Styler with `Styler.use`.  

**Returns**  
`styles` [dict]  

**See also:**  

- **Styler.use** Set the styles on the current Styler.  
- **Styler.copy** Create a copy of the current Styler.

**Notes**

This method is designed to copy non-data dependent attributes of one Styler to another. It differs from `Styler.copy` where data and data dependent attributes are also copied.

The following items are exported since they are not generally data dependent:

- Styling functions added by the `apply` and `applymap`
- Whether axes and names are hidden from the display, if unambiguous.
- Table attributes
- Table styles

The following attributes are considered data dependent and therefore not exported:

- Caption
- UUID
- Tooltips
- Any hidden rows or columns identified by Index labels
- Any formatting applied using `Styler.format`
- Any CSS classes added using `Styler.set_td_classes`

**Examples**

```python
>>> styler = DataFrame([[1, 2], [3, 4]]).style
>>> styler2 = DataFrame([[9, 9, 9]]).style
>>> styler.hide(axis=0).highlight_max(axis=1)
>>> export = styler.export()
>>> styler2.use(export)
```
Styler.format( formatter=None, subset=None, na_rep=None, precision=None, decimal=',', thousands=None, escape=None, hyperlinks=None)

Format the text display value of cells.

Parameters

- **formatter** [str, callable, dict or None] Object to define how values are displayed. See notes.
- **subset** [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[:, <subset>] where the columns are prioritised, to limit data to before applying the function.
- **na_rep** [str, optional] Representation for missing values. If na_rep is None, no special formatting is applied.
  New in version 1.0.0.
- **precision** [int, optional] Floating point precision to use for display purposes, if not determined by the specified formatter.
  New in version 1.3.0.
- **decimal** [str, default "]"] Character used as decimal separator for floats, complex and integers.
  New in version 1.3.0.
- **thousands** [str, optional, default None] Character used as thousands separator for floats, complex and integers.
  New in version 1.3.0.
- **escape** [str, optional] Use ‘html’ to replace the characters &, <, >, ', and " in cell display string with HTML-safe sequences. Use ‘latex’ to replace the characters \, $, \#, _, \{, \}, \~, \^, and \ in the cell display string with LaTeX-safe sequences. Escaping is done before formatter.
  New in version 1.3.0.
- **hyperlinks** ["html", "latex"], optional] Convert string patterns containing https://, http://, ftp:// or www. to HTML <a> tags as clickable URL hyperlinks if “html”, or LaTeX href commands if “latex”.
  New in version 1.4.0.

Returns

- **self** [Styler]

See also:

Styler.format_index Format the text display value of index labels.
Notes

This method assigns a formatting function, \texttt{formatter}, to each cell in the DataFrame. If \texttt{formatter} is \texttt{None}, then the default formatter is used. If a callable then that function should take a data value as input and return a displayable representation, such as a string. If \texttt{formatter} is given as a string this is assumed to be a valid Python format specification and is wrapped to a callable as \texttt{string.format(x)}. If a \texttt{dict} is given, keys should correspond to column names, and values should be string or callable, as above.

The default formatter currently expresses floats and complex numbers with the pandas display precision unless using the \texttt{precision} argument here. The default formatter does not adjust the representation of missing values unless the \texttt{na_rep} argument is used.

The subset argument defines which region to apply the formatting function to. If the \texttt{formatter} argument is given in \texttt{dict} form but does not include all columns within the subset then these columns will have the default formatter applied. Any columns in the formatter dict excluded from the subset will be ignored.

When using a \texttt{formatter} string the dtypes must be compatible, otherwise a \texttt{ValueError} will be raised.

When instantiating a Styler, default formatting can be applied by setting the \texttt{pandas.options}:

\begin{itemize}
  \item \texttt{styler.format.formatter}: default \texttt{None}.
  \item \texttt{styler.format.na_rep}: default \texttt{None}.
  \item \texttt{styler.format.precision}: default \texttt{6}.
  \item \texttt{styler.format.decimal}: default \texttt{"."}.
  \item \texttt{styler.format.thousands}: default \texttt{None}.
  \item \texttt{styler.format.escape}: default \texttt{None}.
\end{itemize}

\begin{warning}
\texttt{Styler.format} is ignored when using the output format \texttt{Styler.to_excel}, since Excel and Python have inherently different formatting structures. However, it is possible to use the \texttt{number-format} pseudo CSS attribute to force Excel permissible formatting. See examples.
\end{warning}

Examples

Using \texttt{na_rep} and \texttt{precision} with the default formatter

\begin{verbatim}
>>> df = pd.DataFrame([[np.nan, 1.0, 'A'], [2.0, np.nan, 3.0]])
>>> df.style.format(na_rep='MISS', precision=3)
     0     1     2
0  MISS  1.000  A
1  2.000  MISS  3.000
\end{verbatim}

Using a \texttt{formatter} specification on consistent column dtypes

\begin{verbatim}
>>> df.style.format('{{:.2f}}', na_rep='MISS', subset=[0,1])
     0     1
0  MISS  1.00  A
1  2.00  MISS  3.000000
\end{verbatim}

Using the default \texttt{formatter} for unspecified columns
Pandas defines a number-format pseudo CSS attribute instead of the `format` method to create to_excel permissible formatting. Note that semi-colons are CSS protected characters but used as separators in Excel’s format string. Replace semi-colons with the section separator character (ASCII-245) when defining the formatting here.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> df = pd.DataFrame({'A': [1, 0, -1]})
>>> pseudo_css = "number-format: 0§[Red](0)§-§@;"
>>> df.style.applymap(lambda v: css).to_excel("formatted_file.xlsx")
...
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**pandas.io.formats.style.Styler.format_index**

Styler.format_index(formatter=None, axis=0, level=None, na_rep=None, precision=None, decimal='.', thousands=None, escape=None, hyperlinks=None)

Format the text display value of index labels or column headers.

New in version 1.4.0.

**Parameters**

- **formatter** [str, callable, dict or None] Object to define how values are displayed. See notes.
- **axis** [[0, “index”], 1, “columns”]] Whether to apply the formatter to the index or column headers.
- **level** [int, str, list] The level(s) over which to apply the generic formatter.
- **na_rep** [str, optional] Representation for missing values. If na_rep is None, no special formatting is applied.
- **precision** [int, optional] Floating point precision to use for display purposes, if not determined by the specified formatter.
- **decimal** [str, default ‘.’] Character used as decimal separator for floats, complex and integers.
- **thousands** [str, optional, default None] Character used as thousands separator for floats, complex and integers.
- **escape** [str, optional] Use ‘html’ to replace the characters &lt;, &gt;, ‘ and " in cell display string with HTML-safe sequences. Use ‘latex’ to replace the characters &and%, $, #, _, {}, ~, ^, and \ in the cell display string with LaTeX-safe sequences. Escaping is done before formatter.
- **hyperlinks** [“html”, “latex”], optional] Convert string patterns containing https://, http://, ftp:// or www. to HTML <a> tags as clickable URL hyperlinks if “html”, or LaTeX href commands if “latex”.

**Returns**
self [Styler]

See also:

**Styler.format** Format the text display value of data cells.

Notes

This method assigns a formatting function, `formatter`, to each level label in the DataFrame’s index or column headers. If `formatter` is `None`, then the default formatter is used. If a callable then that function should take a label value as input and return a displayable representation, such as a string. If `formatter` is given as a string this is assumed to be a valid Python format specification and is wrapped to a callable as `str.format(x)`. If a `dict` is given, keys should correspond to MultiIndex level numbers or names, and values should be string or callable, as above.

The default formatter currently expresses floats and complex numbers with the pandas display precision unless using the `precision` argument here. The default formatter does not adjust the representation of missing values unless the `na_rep` argument is used.

The `level` argument defines which levels of a MultiIndex to apply the method to. If the `formatter` argument is given in dict form but does not include all levels within the level argument then these unspecified levels will have the default formatter applied. Any levels in the formatter dict specifically excluded from the level argument will be ignored.

When using a `formatter` string the dtypes must be compatible, otherwise a `ValueError` will be raised.

**Warning:** *Styler.format_index* is ignored when using the output format *Styler.to_excel*, since Excel and Python have inherently different formatting structures. However, it is possible to use the *number-format* pseudo CSS attribute to force Excel permissible formatting. See documentation for *Styler.format*.

Examples

Using `na_rep` and `precision` with the default formatter

```python
>>> df = pd.DataFrame([[1, 2, 3]], columns=[2.0, np.nan, 4.0])
>>> df.style.format_index(axis=1, na_rep='MISS', precision=3)
               2.000   MISS   4.000
0          1    2    3
```

Using a `formatter` specification on consistent dtypes in a level

```python
>>> df.style.format_index('{:.2f}', axis=1, na_rep='MISS')
               2.00   MISS   4.00
0          1    2    3
```

Using the default formatter for unspecified levels

```python
>>> df = pd.DataFrame([[1, 2, 3]],
...                    columns=pd.MultiIndex.from_arrays(["a", "a", "b", [2, np.nan, 4]]))
>>> df.style.format_index({0: lambda v: upper(v), axis=1, precision=1})
```

(continues on next page)
Using a callable formatter function.

```python
>>> func = lambda s: 'STRING' if isinstance(s, str) else 'FLOAT'
>>> df.style.format_index(func, axis=1, na_rep='MISS')
...  STR... STRING
0  2.0  1  3.0
```

Using a formatter with HTML escape and na_rep.

```python
>>> df = pd.DataFrame([[1, 2, 3]], columns=['A', 'A&B', None])
>>> s = df.style.format_index('$ {{0}}', axis=1, escape='html', na_rep='NA')
... <th .. >$ 1</th>
```

Using a formatter with LaTeX escape.

```python
>>> df = pd.DataFrame([[1, 2, 3]], columns=['123', '~', '$%#'])
>>> df.style.format_index('{\textbf{{{}}} }', escape='latex', axis=1).to_latex()
... \begin{tabular}{|l|l|l|l|}
\hline
{} & \textbf{123} & \textbf{~} & \textbf{$\%#$} \\
\hline
0 & 1 & 2 & 3 \\
\hline
\end{tabular}
```

**pandas.io.formats.style.Styler.from_custom_template**

**classmethod** `Styler.from_custom_template(searchpath, html_table=None, html_style=None)`

Factory function for creating a subclass of *Styler*.

Uses custom templates and Jinja environment.

Changed in version 1.3.0.

**Parameters**

- **searchpath** [str or list] Path or paths of directories containing the templates.
- **html_table** [str] Name of your custom template to replace the html_table template.
  
  New in version 1.3.0.
- **html_style** [str] Name of your custom template to replace the html_style template.
  
  New in version 1.3.0.

**Returns**

- **MyStyler** [subclass of *Styler*] Has the correct env,`template_html`, `template_html_table` and `template_html_style` class attributes set.
**pandas.io.formats.style.Styler.hide**

Styler.hide(subset=None, axis=0, level=None, names=False)

Hide the entire index / column headers, or specific rows / columns from display.

New in version 1.4.0.

**Parameters**

- **subset** [label, array-like, IndexSlice, optional] A valid 1d input or single key along the axis within DataFrame.loc[<subset>, :] or DataFrame.loc[:, <subset>] depending upon axis, to limit data to select hidden rows / columns.

- **axis** [(“index”, 0, “columns”, 1)] Apply to the index or columns.

- **level** [int, str, list] The level(s) to hide in a MultiIndex if hiding the entire index / column headers. Cannot be used simultaneously with subset.

- **names** [bool] Whether to hide the level name(s) of the index / columns headers in the case it (or at least one the levels) remains visible.

**Returns**

self [Styler]

**Notes**

This method has multiple functionality depending upon the combination of the subset, level and names arguments (see examples). The axis argument is used only to control whether the method is applied to row or column headers:

<table>
<thead>
<tr>
<th>subset</th>
<th>level</th>
<th>names</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None</td>
<td>False</td>
<td>The axis-Index is hidden entirely.</td>
</tr>
<tr>
<td>None</td>
<td>None</td>
<td>True</td>
<td>Only the axis-Index names are hidden.</td>
</tr>
<tr>
<td>None</td>
<td>Int, Str, List</td>
<td>False</td>
<td>Specified axis-MultiIndex levels are hidden entirely.</td>
</tr>
<tr>
<td>None</td>
<td>Int, Str, List</td>
<td>True</td>
<td>Specified axis-MultiIndex levels are hidden entirely and the names of remaining axis-MultiIndex levels.</td>
</tr>
<tr>
<td>Subset</td>
<td>None</td>
<td>False</td>
<td>The specified data rows/columns are hidden, but the axis-Index itself, and names, remain unchanged.</td>
</tr>
<tr>
<td>Subset</td>
<td>None</td>
<td>True</td>
<td>The specified data rows/columns and axis-Index names are hidden, but the axis-Index itself remains unchanged.</td>
</tr>
<tr>
<td>Subset</td>
<td>Int, Str, List</td>
<td>Boolean</td>
<td>ValueError: cannot supply subset and level simultaneously.</td>
</tr>
</tbody>
</table>

Note this method only hides the identified elements so can be chained to hide multiple elements in sequence.
Examples

Simple application hiding specific rows:

```python
>>> df = pd.DataFrame([[1,2], [3,4], [5,6]], index=["a", "b", "c"])
>>> df.style.hide(["a", "b"])
0   1
  c  5  6
```

Hide the index and retain the data values:

```python
>>> midx = pd.MultiIndex.from_product(["x", "y"], ["a", "b", "c"])
>>> df = pd.DataFrame(np.random.randn(6,6), index=midx, columns=midx)
>>> df.style.format("{:.1f}").hide()
```

Hide specific rows in a MultiIndex but retain the index:

```python
>>> df.style.format("{:.1f}").hide(subset=(slice(None), ["a", "c"]))
...```

Hide specific rows and the index through chaining:

```python
>>> df.style.format("{:.1f}").hide(subset=(slice(None), ["a", "c"])).hide()
...```

Hide a specific level:

```python
>>> df.style.format("{:.1f}").hide(level=1)
```

Hiding just the index level names:
Examples all produce equivalently transposed effects with `axis="columns"`.  

**pandas.io.formats.style.Styler.hide_columns**

`Styler.hide_columns`(subset=None, level=None, names=False)

Hide the column headers or specific keys in the columns from rendering.

This method has dual functionality:

- if `subset` is `None` then the entire column headers row, or specific levels, will be hidden whilst the data-values remain visible.
- if a `subset` is given then those specific columns, including the data-values will be hidden, whilst the column headers row remains visible.

Changed in version 1.3.0.

.. deprecated:: 1.4.0  This method should be replaced by `hide(axis="columns", **kwargs)`

**Parameters**

- **subset** [label, array-like, IndexSlice, optional] A valid 1d input or single key along the columns axis within `DataFrame.loc[:,<subset>]`, to limit data to before applying the function.
- **level** [int, str, list] The level(s) to hide in a MultiIndex if hiding the entire column headers row. Cannot be used simultaneously with `subset`.

New in version 1.4.0.
- **names** [bool] Whether to hide the column index name(s), in the case all column headers, or some levels, are visible.

New in version 1.4.0.

**Returns**

- **self** [Styler]

**See also:**

**Styler.hide** Hide the entire index / columns, or specific rows / columns.
Styler.hide_index

Styler.hide_index(subset=None, level=None, names=False)

Hide the entire index, or specific keys in the index from rendering.

This method has dual functionality:

- if `subset` is `None` then the entire index, or specified levels, will be hidden whilst displaying all data-rows.
- if a `subset` is given then those specific rows will be hidden whilst the index itself remains visible.

Changed in version 1.3.0.

Deprecated since version 1.4.0: This method should be replaced by `hide(axis="index", **kwargs)

Parameters

- `subset` [label, array-like, IndexSlice, optional] A valid 1d input or single key along the index axis within `DataFrame.loc[<subset>, :]`, to limit data to before applying the function.
- `level` [int, str, list] The level(s) to hide in a MultiIndex if hiding the entire index. Cannot be used simultaneously with `subset`.
  
  New in version 1.4.0.
- `names` [bool] Whether to hide the index name(s), in the case the index or part of it remains visible.
  
  New in version 1.4.0.

Returns

- `self` [Styler]

See also:

Styler.hide  Hide the entire index / columns, or specific rows / columns.

Styler.highlight_between

Styler.highlight_between(subset=None, color='yellow', axis=0, left=None, right=None, inclusive='both', props=None)

Highlight a defined range with a style.

New in version 1.3.0.

Parameters

- `subset` [label, array-like, IndexSlice, optional] A valid 2d input to `DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to `DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to before applying the function.
- `color` [str, default ‘yellow’] Background color to use for highlighting.
- `axis` [[0 or ‘index’, 1 or ‘columns’, None], default 0] If `left` or `right` given as sequence, axis along which to apply those boundaries. See examples.
- `left` [scalar or datetime-like, or sequence or array-like, default None] Left bound for defining the range.
- `right` [scalar or datetime-like, or sequence or array-like, default None] Right bound for defining the range.
- `inclusive` [‘left’, ‘right’, ‘both’, default ‘both’] Whether to include the left or right boundary in the selection.

New in version 1.3.0.

New in version 1.4.0.
right [scalar or datetime-like, or sequence or array-like, default None] Right bound for defining the range.

inclusive [\{'both', 'neither', 'left', 'right'\}] Identify whether bounds are closed or open.

props [str, default None] CSS properties to use for highlighting. If props is given, color is not used.

Returns

self [Styler]

See also:

Styler.highlight_null Highlight missing values with a style.
Styler.highlight_max Highlight the maximum with a style.
Styler.highlight_min Highlight the minimum with a style.
Styler.highlight_quantile Highlight values defined by a quantile with a style.

Notes

If left is None only the right bound is applied. If right is None only the left bound is applied. If both are None all values are highlighted.

axis is only needed if left or right are provided as a sequence or an array-like object for aligning the shapes. If left and right are both scalars then all axis inputs will give the same result.

This function only works with compatible dtypes. For example a datetime-like region can only use equivalent datetime-like left and right arguments. Use subset to control regions which have multiple dtypes.

Examples

Basic usage

>>> df = pd.DataFrame({
... 'One': [1.2, 1.6, 1.5],
... 'Two': [2.9, 2.1, 2.5],
... 'Three': [3.1, 3.2, 3.8],
... })
>>> df.style.highlight_between(left=2.1, right=2.9)

<table>
<thead>
<tr>
<th></th>
<th>One</th>
<th>Two</th>
<th>Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.2</td>
<td>2.9</td>
<td>3.1</td>
</tr>
<tr>
<td>1</td>
<td>1.6</td>
<td>2.1</td>
<td>3.2</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>2.5</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Using a range input sequence along an axis, in this case setting a left and right for each column individually

>>> df.style.highlight_between(left=[1.4, 2.4, 3.4], right=[1.6, 2.6, 3.6],
... axis=1, color="#ffee75")
Using `axis=None` and providing the `left` argument as an array that matches the input DataFrame, with a constant right

```python
>>> df.style.highlight_between(left=[[2,2,3],[2,2,3],[3,3,3]], right=3.5,
...   axis=None, color="#fffd75")
```

Using `props` instead of default background coloring

```python
>>> df.style.highlight_between(left=1.5, right=3.5,
...   props='font-weight:bold;color:#e83e8c')
```

**pandas.io.formats.style.Styler.highlight_max**

Styler.highlight_max(subset=None, color='yellow', axis=0, props=None)

Highlight the maximum with a style.

**Parameters**

- **subset** [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[, <subset>] where the columns are prioritised, to limit data to before applying the function.

- **color** [str, default ‘yellow’] Background color to use for highlighting.

- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default 0] Apply to each column (axis=0 or 'index'), to each row (axis=1 or 'columns'), or to the entire DataFrame at once with axis=None.

- **props** [str, default None] CSS properties to use for highlighting. If props is given, color is not used.

**Returns**

New in version 1.3.0.
self [Styler]

See also:

**Styler.highlight_null** Highlight missing values with a style.

**Styler.highlight_min** Highlight the minimum with a style.

**Styler.highlight_between** Highlight a defined range with a style.

**Styler.highlight_quantile** Highlight values defined by a quantile with a style.

**pandas.io.formats.style.Styler.highlight_min**

**Styler.highlight_min**(*subset=None, color='yellow', axis=0, props=None*)

Highlight the minimum with a style.

**Parameters**

* subset [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[:, <subset>] where the columns are prioritised, to limit data to before applying the function.

* color [str, default 'yellow'] Background color to use for highlighting.

* axis ([0 or 'index', 1 or 'columns', None], default 0) Apply to each column (axis=0 or 'index'), to each row (axis=1 or 'columns'), or to the entire DataFrame at once with axis=None.

* props [str, default None] CSS properties to use for highlighting. If props is given, color is not used.

**New in version 1.3.0.**

**Returns**

self [Styler]

See also:

**Styler.highlight_null** Highlight missing values with a style.

**Styler.highlight_max** Highlight the maximum with a style.

**Styler.highlight_between** Highlight a defined range with a style.

**Styler.highlight_quantile** Highlight values defined by a quantile with a style.

**pandas.io.formats.style.Styler.highlight_null**

**Styler.highlight_null**(*null_color='red', subset=None, props=None*)

Highlight missing values with a style.

**Parameters**

* null_color [str, default 'red']
**subset** [label, array-like, IndexSlice, optional] A valid 2d input to 
`DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to 
`DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to 
before applying the function.

New in version 1.1.0.

**props** [str, default None] CSS properties to use for highlighting. If props is given, 
`color` is not used.

New in version 1.3.0.

**Returns**

* self [Styler]

**See also:**

* `Styler.highlight_max` Highlight the maximum with a style.
* `Styler.highlight_min` Highlight the minimum with a style.
* `Styler.highlight_between` Highlight a defined range with a style.
* `Styler.highlight_quantile` Highlight values defined by a quantile with a style.

**pandas.io.formats.style.Styler.highlight_quantile**

`Styler.highlight_quantile(subset=None, color='yellow', axis=0, q_left=0.0, q_right=1.0, 
interpolation='linear', inclusive='both', props=None)`

Highlight values defined by a quantile with a style.

New in version 1.3.0.

**Parameters**

* **subset** [label, array-like, IndexSlice, optional] A valid 2d input to 
  `DataFrame.loc[<subset>]`, or, in the case of a 1d input or single key, to 
  `DataFrame.loc[:, <subset>]` where the columns are prioritised, to limit data to 
  before applying the function.

* **color** [str, default 'yellow'] Background color to use for highlighting.

* **axis** [[0 or ‘index’, 1 or ‘columns’, None], default 0] Axis along which to determine 
  and highlight quantiles. If None quantiles are measured over the entire DataFrame. 
  See examples.

* **q_left** [float, default 0] Left bound, in [0, q_right), for the target quantile range.

* **q_right** [float, default 1] Right bound, in (q_left, 1], for the target quantile range.

  `Series.quantile` or `DataFrame.quantile` for quantile estimation.

* **inclusive** [[‘both’, ‘neither’, ‘left’, ‘right’]] Identify whether quantile bounds are closed 
  or open.

* **props** [str, default None] CSS properties to use for highlighting. If props is given, 
  `color` is not used.

**Returns**

* self [Styler]
See also:

- **Styler.highlight_null** Highlight missing values with a style.
- **Styler.highlight_max** Highlight the maximum with a style.
- **Styler.highlight_min** Highlight the minimum with a style.
- **Styler.highlight_between** Highlight a defined range with a style.

**Notes**

This function does not work with `str` dtypes.

**Examples**

Using `axis=None` and apply a quantile to all collective data

```python
>>> df = pd.DataFrame(np.arange(10).reshape(2,5) + 1)
>>> df.style.highlight_quantile(axis=None, q_left=0.8, color="#fffd75")
```

```
   0  1  2  3  4
0  0  1  2  3  4  5
1  6  7  8  9 10
```

Or highlight quantiles row-wise or column-wise, in this case by row-wise

```python
>>> df.style.highlight_quantile(axis=1, q_left=0.8, color="#fffd75")
```

```
   0  1  2  3  4
0  0  1  2  3  4  5
1  6  7  8  9 10
```

Use `props` instead of default background coloring

```python
>>> df.style.highlight_quantile(axis=None, q_left=0.2, q_right=0.8,
...                             props='font-weight:bold;color:#e83e8c')
```

```
   0  1  2  3  4
0  0  1  2  3  4  5
1  6  7  8  9 10
```
**Styler.pipe**

*Styler.pipe*(func, *args, **kwargs)

Apply func(self, *args, **kwargs), and return the result.

**Parameters**

- `func` [function] Function to apply to the Styler. Alternatively, a (callable, keyword) tuple where keyword is a string indicating the keyword of callable that expects the Styler.
- `*args` [optional] Arguments passed to func.
- `**kwargs` [optional] A dictionary of keyword arguments passed into func.

**Returns**

- `object` : The value returned by func.

**See also:**

- **DataFrame.pipe** An analogous method for DataFrame.
- **Styler.apply** Apply a CSS-styling function column-wise, row-wise, or table-wise.

**Notes**

Like DataFrame.pipe(), this method can simplify the application of several user-defined functions to a styler. Instead of writing:

\[
\text{f(g(df.style.set_precision(3), arg1=a), arg2=b, arg3=c)}
\]

users can write:

\[
(df.style.set_precision(3)
   .pipe(g, arg1=a)
   .pipe(f, arg2=b, arg3=c))
\]

In particular, this allows users to define functions that take a styler object, along with other parameters, and return the styler after making styling changes (such as calling **Styler.apply()** or **Styler.set_properties()**). Using .pipe, these user-defined style “transformations” can be interleaved with calls to the built-in Styler interface.

**Examples**

```python
>>> def format_conversion(styler):
...     return (styler.set_properties(**{\'text-align\': \'right\'}))
...     .format({'conversion': '{:.1%}'}))
```

The user-defined `format_conversion` function above can be called within a sequence of other style modifications:

```python
>>> df = pd.DataFrame({'trial': list(range(5)),
...                     'conversion': [0.75, 0.85, np.nan, 0.7, 0.72]})
>>> (df.style
```

(continues on next page)
... .highlight_min(subset=['conversion'], color='yellow')
... .pipe(format_conversion)
... .set_caption("Results with minimum conversion highlighted.\))
...

pandas.io.formats.style.Styler.render

Styler.render(sparse_index=None, sparse_columns=None, **kwargs)

Render the Styler including all applied styles to HTML.

Deprecated since version 1.4.0.

Parameters

- **sparse_index** [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. Defaults to pandas.options.styler.sparse.index value.

- **sparse_columns** [bool, optional] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. Defaults to pandas.options.styler.sparse.columns value.

- **kwargs** Any additional keyword arguments are passed through to self.template.render. This is useful when you need to provide additional variables for a custom template.

Returns

- **rendered** [str] The rendered HTML.
Notes

This method is deprecated in favour of Styler.to_html.

Styler objects have defined the _repr_html_ method which automatically calls self.to_html() when it’s the last item in a Notebook cell.

When calling Styler.render() directly, wrap the result in IPython.display.HTML to view the rendered HTML in the notebook.

Pandas uses the following keys in render. Arguments passed in **kwargs take precedence, so think carefully if you want to override them:

- head
- cellstyle
- body
- uuid
- table_styles
- caption
- table_attributes

pandas.io.formats.style.Styler.set_caption

Styler.set_caption(caption)

Set the text added to a <caption> HTML element.

Parameters

caption [str, tuple] For HTML output either the string input is used or the first element of the tuple. For LaTeX the string input provides a caption and the additional tuple input allows for full captions and short captions, in that order.

Returns

self [Styler]

pandas.io.formats.style.Styler.set_na_rep

Styler.set_na_rep(na_rep)

Set the missing data representation on a Styler.

New in version 1.0.0.

Deprecated since version 1.3.0.

Parameters

na_rep [str]

Returns

self [Styler]
Notes

This method is deprecated. See `Styler.format()`

**pandas.io.formats.style.Styler.set_precision**

Styler.set_precision(\texttt{precision})

Set the precision used to display values.

Deprecated since version 1.3.0.

Parameters

- \texttt{precision} \texttt{[int]}

Returns

- self \texttt{[Styler]}

Notes

This method is deprecated see `Styler.format()

**pandas.io.formats.style.Styler.set_properties**

Styler.set_properties(\texttt{subset=\texttt{None}}, **\texttt{kwargs}**)

Set defined CSS-properties to each \texttt{\langle td\rangle} HTML element within the given subset.

Parameters

- \texttt{subset} \texttt{[label, array-like, IndexSlice, optional]} A valid 2d input to \texttt{DataFrame.loc[\langle subset\rangle]}, or, in the case of a 1d input or single key, to \texttt{DataFrame.loc[:, \langle subset\rangle]} where the columns are prioritised, to limit data to before applying the function.

- **\texttt{kwargs}** \texttt{[dict]} A dictionary of property, value pairs to be set for each cell.

Returns

- self \texttt{[Styler]}

Notes

This is a convenience methods which wraps the `Styler.applymap()` calling a function returning the CSS-properties independently of the data.
Examples

```python
def = pd.DataFrame(np.random.randn(10, 4))
def.style.set_properties(color="white", align="right")
def.style.set_properties(**{"background-color": 'yellow'})
```

See Table Visualization user guide for more details.

`pandas.io.formats.style.Styler.set_sticky`

`Styler.set_sticky(axis=0, pixel_size=None, levels=None)`

Add CSS to permanently display the index or column headers in a scrolling frame.

**Parameters**

- **axis** [0 or 'index', 1 or 'columns', default 0] Whether to make the index or column headers sticky.
- **pixel_size** [int, optional] Required to configure the width of index cells or the height of column header cells when sticking a MultiIndex (or with a named Index). Defaults to 75 and 25 respectively.
- **levels** [int, str, list, optional] If `axis` is a MultiIndex the specific levels to stick. If `None` will stick all levels.

**Returns**

self [Styler]

**Notes**

This method uses the CSS ‘position: sticky;’ property to display. It is designed to work with visible axes, therefore both:

- `styler.set_sticky(axis="index").hide(axis="index")`
- `styler.set_sticky(axis="columns").hide(axis="columns")`

may produce strange behaviour due to CSS controls with missing elements.

`pandas.io.formats.style.Styler.set_table_attributes`

`Styler.set_table_attributes(attributes)`

Set the table attributes added to the `<table>` HTML element. These are items in addition to automatic (by default) id attribute.

**Parameters**

- **attributes** [str]

**Returns**

self [Styler]

See also:

- `Styler.set_table_styles` Set the table styles included within the `<style>` HTML element.
**Styler.set_td_classes**  Set the DataFrame of strings added to the class attribute of `<td>` HTML elements.

**Examples**

```python
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_attributes('class="pure-table"')
# ... <table class="pure-table"> ...
```

**pandas.io.formats.style.Styler.set_table_styles**

**Styler.set_table_styles**(table_styles=None, axis=0, overwrite=True, css_class_names=None)

Set the table styles included within the `<style>` HTML element.

This function can be used to style the entire table, columns, rows or specific HTML selectors.

**Parameters**

- **table_styles** [list or dict] If supplying a list, each individual table_style should be a dictionary with selector and props keys. selector should be a CSS selector that the style will be applied to (automatically prefixed by the table’s UUID) and props should be a list of tuples with (attribute, value). If supplying a dict, the dict keys should correspond to column names or index values, depending upon the specified axis argument. These will be mapped to row or col CSS selectors. MultiIndex values as dict keys should be in their respective tuple form. The dict values should be a list as specified in the form with CSS selectors and props that will be applied to the specified row or column.

  Changed in version 1.2.0.

- **axis** [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (axis=0 or 'index'), to each row (axis=1 or 'columns'). Only used if table_styles is dict.

  New in version 1.2.0.

- **overwrite** [bool, default True] Styles are replaced if True, or extended if False. CSS rules are preserved so most recent styles set will dominate if selectors intersect.

  New in version 1.2.0.

- **css_class_names** [dict, optional] A dict of strings used to replace the default CSS classes described below.

  New in version 1.4.0.

**Returns**

- **self** [Styler]

**See also:**

- **Styler.set_td_classes** Set the DataFrame of strings added to the class attribute of `<td>` HTML elements.

- **Styler.set_table_attributes** Set the table attributes added to the `<table>` HTML element.
Notes

The default CSS classes dict, whose values can be replaced is as follows:

```python
css_class_names = {
    "row_heading": "row_heading",
    "col_heading": "col_heading",
    "index_name": "index_name",
    "col": "col",
    "col_trim": "col_trim",
    "row_trim": "row_trim",
    "level": "level",
    "data": "data",
    "blank": "blank"
}
```

Examples

```python
>>> df = pd.DataFrame(np.random.randn(10, 4),
...                   columns=['A', 'B', 'C', 'D'])

>>> df.style.set_table_styles(
...     [{'selector': 'tr:hover',
...       'props': [{'background-color': 'yellow'}]}]
... )
```

Or with CSS strings

```python
>>> df.style.set_table_styles(
...     [{'selector': 'tr:hover',
...       'props': 'background-color: yellow; font-size: 1em;'}]
... )
```

Adding column styling by name

```python
>>> df.style.set_table_styles(
...     {'A': [{'selector': '',
...            'props': [{'color': 'red'}]},
...            'B': [{'selector': 'td',
...                  'props': 'color: blue;'}]
...     }, overwrite=False)
```

Adding row styling

```python
>>> df.style.set_table_styles(
...     {0: [{'selector': 'td:hover',
...          'props': [{'font-size': '25px'}]}]
... }, axis=1, overwrite=False)
```

See Table Visualization user guide for more details.
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.io.formats.style.Styler.set_td_classes

**Styler.set_td_classes(classes)**

Set the DataFrame of strings added to the class attribute of <td> HTML elements.

**Parameters**

- **classes** [DataFrame] DataFrame containing strings that will be translated to CSS classes, mapped by identical column and index key values that must exist on the underlying Styler data. None, NaN values, and empty strings will be ignored and not affect the rendered HTML.

**Returns**

- **self** [Styler]

See also:

- **Styler.set_table_styles** Set the table styles included within the <style> HTML element.
- **Styler.set_table_attributes** Set the table attributes added to the <table> HTML element.

**Notes**

Can be used in combination with Styler.set_table_styles to define an internal CSS solution without reference to external CSS files.

**Examples**

```python
>>> df = pd.DataFrame(data=[[1, 2, 3], [4, 5, 6]], columns=["A", "B", "C"], index=["index1", "index2"]
>>> classes = pd.DataFrame(
...     [["min-val red", ""],
...      ["red", "blue max-val"]],
...     index=df.index, columns=df.columns)
>>> df.style.set_td_classes(classes)
```

Using MultiIndex columns and a classes DataFrame as a subset of the underlying.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], index=["a", "b"],
...     columns=["level0", "level0", "levella", "level0b"])
>>> classes = pd.DataFrame(["min-val"], index=["a"],
...     columns=["level0","levella"])
>>> df.style.set_td_classes(classes)
```

Form of the output with new additional css classes,

```python
>>> df = pd.DataFrame([[1]])
>>> css = pd.DataFrame(["other-class"], index=["a"],
...     columns=["other-class"])
>>> s = Styler(df, uuid="_", cell_ids=False).set_td_classes(css)
>>> s.hide(axis=0).to_html()
'"<style type="text/css"></style>"'
'"<table id="T_" >"'
'"<thead>
'"<tr><th class="col_heading level0 col0" >0</th></tr>
```

(continues on next page)
pandas.io.formats.style.Styler.set_tooltips

Styler.set_tooltips(ttips, props=None, css_class=None)
Set the DataFrame of strings on Styler generating :hover tooltips.
These string based tooltips are only applicable to <td> HTML elements, and cannot be used for column or index headers.
New in version 1.3.0.

Parameters

- **ttips** [DataFrame] DataFrame containing strings that will be translated to tooltips, mapped by identical column and index values that must exist on the underlying Styler data. None, NaN values, and empty strings will be ignored and not affect the rendered HTML.

- **props** [list-like or str, optional] List of (attr, value) tuples or a valid CSS string. If None adopts the internal default values described in notes.

- **css_class** [str, optional] Name of the tooltip class used in CSS, should conform to HTML standards. Only useful if integrating tooltips with external CSS. If None uses the internal default value 'pd-t'.

Returns

- **self** [Styler]

Notes

Tooltips are created by adding `<span class="pd-t"></span>` to each data cell and then manipulating the table level CSS to attach pseudo hover and pseudo after selectors to produce the required results.

The default properties for the tooltip CSS class are:

- visibility: hidden
- position: absolute
- z-index: 1
- background-color: black
- color: white
- transform: translate(-20px, -20px)

The property ‘visibility: hidden;’ is a key prerequisite to the hover functionality, and should always be included in any manual properties specification, using the props argument.

Tooltips are not designed to be efficient, and can add large amounts of additional HTML for larger tables, since they also require that cell_ids is forced to True.
Examples

Basic application

```python
>>> df = pd.DataFrame(data=[[0, 1], [2, 3]])
>>> ttips = pd.DataFrame(
    ...     data=[["Min", ""], [np.nan, "Max"]], columns=df.columns, index=df.index
    ... )
>>> s = df.style.set_tooltips(ttips).to_html()
```

Optionally controlling the tooltip visual display

```python
>>> df.style.set_tooltips(ttips, css_class='tt-add', props=[
    ...     ('visibility', 'hidden'),
    ...     ('position', 'absolute'),
    ...     ('z-index', 1))
>>> df.style.set_tooltips(ttips, css_class='tt-add',
    ...     props='visibility:hidden; position:absolute; z-index:1;')
```

```
pandas.io.formats.style.Styler.set_uuid
```

**Styler.set_uuid(uuid)**

Set the uuid applied to id attributes of HTML elements.

**Parameters**

- **uuid** [str]

**Returns**

- **self** [Styler]

**Notes**

Almost all HTML elements within the table, and including the `<table>` element are assigned id attributes. The format is T_uuid_<extra> where <extra> is typically a more specific identifier, such as row1_col2.

```
pandas.io.formats.style.Styler.text_gradient
```

**Styler.text_gradient(cmap='PuBu', low=0, high=0, axis=0, subset=None, vmin=None, vmax=None, gmap=None)**

Color the text in a gradient style.

The text color is determined according to the data in each column, row or frame, or by a given gradient map. Requires matplotlib.

**Parameters**

- **cmap** [str or colormap] Matplotlib colormap.
- **low** [float] Compress the color range at the low end. This is a multiple of the data range to extend below the minimum; good values usually in [0, 1], defaults to 0.
**High** [float] Compress the color range at the high end. This is a multiple of the data range to extend above the maximum; good values usually in [0, 1], defaults to 0.

**Axis** [[0 or 'index', 1 or 'columns', None], default 0] Apply to each column (axis=0 or 'index'), to each row (axis=1 or 'columns'), or to the entire DataFrame at once with axis=None.

**Subset** [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[:, <subset>] where the columns are prioritised, to limit data before applying the function.

**Text Color Threshold** [float or int] This argument is ignored (only used in background_gradient). Luminance threshold for determining text color in [0, 1]. Facilitates text visibility across varying background colors. All text is dark if 0, and light if 1, defaults to 0.408.

**Vmin** [float, optional] Minimum data value that corresponds to colormap minimum value. If not specified the minimum value of the data (or gmap) will be used.

New in version 1.0.0.

**Vmax** [float, optional] Maximum data value that corresponds to colormap maximum value. If not specified the maximum value of the data (or gmap) will be used.

New in version 1.0.0.

**Gmap** [array-like, optional] Gradient map for determining the text colors. If not supplied will use the underlying data from rows, columns or frame. If given as an ndarray or list-like must be an identical shape to the underlying data considering axis and subset. If given as DataFrame or Series must have same index and column labels considering axis and subset. If supplied, vmin and vmax should be given relative to this gradient map.

New in version 1.3.0.

**Returns**

**Self** [Styler]

**See Also:**

Styler.background_gradient Color the background in a gradient style.

**Notes**

When using low and high the range of the gradient, given by the data if gmap is not given or by gmap, is extended at the low end effectively by map.min - low * map.range and at the high end by map.max + high * map.range before the colors are normalized and determined.

If combining with vmin and vmax the map.min, map.max and map.range are replaced by values according to the values derived from vmin and vmax.

This method will preselect numeric columns and ignore non-numeric columns unless a gmap is supplied in which case no preselection occurs.
Examples

```python
>>> df = pd.DataFrame(columns=['City', 'Temp (c)', 'Rain (mm)', 'Wind (m/s)'],
                   data=[['Stockholm', 21.6, 5.0, 3.2],
                         ['Oslo', 22.4, 13.3, 3.1],
                         ['Copenhagen', 24.5, 0.0, 6.7]])
```

Shading the values column-wise, with `axis=0`, preselecting numeric columns

```python
>>> df.style.text_gradient(axis=0)
```

<table>
<thead>
<tr>
<th>City</th>
<th>Temp (c)</th>
<th>Rain (mm)</th>
<th>Wind (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stockholm</td>
<td>21.60000</td>
<td>5.00000</td>
<td>3.20000</td>
</tr>
<tr>
<td>Oslo</td>
<td>22.40000</td>
<td>13.30000</td>
<td>3.10000</td>
</tr>
<tr>
<td>Copenhagen</td>
<td>24.50000</td>
<td>0.00000</td>
<td>6.70000</td>
</tr>
</tbody>
</table>

Shading all values collectively using `axis=None`

```python
>>> df.style.text_gradient(axis=None)
```

<table>
<thead>
<tr>
<th>City</th>
<th>Temp (c)</th>
<th>Rain (mm)</th>
<th>Wind (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stockholm</td>
<td>21.60000</td>
<td>5.00000</td>
<td>3.20000</td>
</tr>
<tr>
<td>Oslo</td>
<td>22.40000</td>
<td>13.30000</td>
<td>3.10000</td>
</tr>
<tr>
<td>Copenhagen</td>
<td>24.50000</td>
<td>0.00000</td>
<td>6.70000</td>
</tr>
</tbody>
</table>

Compress the color map from the both low and high ends

```python
>>> df.style.text_gradient(axis=None, low=0.75, high=1.0)
```

<table>
<thead>
<tr>
<th>City</th>
<th>Temp (c)</th>
<th>Rain (mm)</th>
<th>Wind (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stockholm</td>
<td>21.60000</td>
<td>5.00000</td>
<td>3.20000</td>
</tr>
<tr>
<td>Oslo</td>
<td>22.40000</td>
<td>13.30000</td>
<td>3.10000</td>
</tr>
<tr>
<td>Copenhagen</td>
<td>24.50000</td>
<td>0.00000</td>
<td>6.70000</td>
</tr>
</tbody>
</table>

Manually setting vmin and vmax gradient thresholds

```python
>>> df.style.text_gradient(axis=None, vmin=6.7, vmax=21.6)
```

Setting a gmap and applying to all columns with another cmap

```python
>>> df.style.text_gradient(axis=0, gmap=df['Temp (c)'], cmap='YlOrRd')
...```

Setting the gradient map for a dataframe (i.e. `axis=None`), we need to explicitly state subset to match the gmap shape
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
>>> gmap = np.array([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
>>> df.style.text_gradient(axis=None, gmap=gmap,
... cmap='YlOrRd', subset=['Temp (c)', 'Rain (mm)', 'Wind (m/s)'])

<table>
<thead>
<tr>
<th>City</th>
<th>Temp (c)</th>
<th>Rain (mm)</th>
<th>Wind (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Stockholm</td>
<td>21.600000</td>
<td>5.000000</td>
</tr>
<tr>
<td>1</td>
<td>Oslo</td>
<td>22.400000</td>
<td>13.300000</td>
</tr>
<tr>
<td>2</td>
<td>Copenhagen</td>
<td>24.500000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
```

**pandas.io.formats.style.Styler.use**

**Styler.use(**styles**)

Set the styles on the current Styler.

Possibly uses styles from *Styler.export*.

**Parameters**

- **styles** [dict(str, Any)]

**List of attributes to add to Styler. Dict keys should contain only:**

- “apply”: list of styler functions, typically added with *apply* or *applymap*.
- “table_attributes”: HTML attributes, typically added with *set_table_attributes*.
- “table_styles”: CSS selectors and properties, typically added with *set_table_styles*.
- “hide_index”: whether the index is hidden, typically added with *hide_index*, or a boolean list for hidden levels.
- “hide_columns”: whether column headers are hidden, typically added with *hide_columns*, or a boolean list for hidden levels.
- “hide_index_names”: whether index names are hidden.
- “hide_column_names”: whether column header names are hidden.
• “css”: the css class names used.

Returns

self [Styler]

See also:

Styler.export Export the non data dependent attributes to the current Styler.

Examples

```python
>>> styler = DataFrame([[1, 2], [3, 4]]).style
>>> styler2 = DataFrame([[9, 9, 9]]).style
>>> styler.hide(axis=0).highlight_max(axis=1)
>>> export = styler.export()
>>> styler2.use(export)
```

pandas.io.formats.style.Styler.where

Styler.where(cond, value, other=None, subset=None, **kwargs)

Apply CSS-styles based on a conditional function elementwise.

Deprecated since version 1.3.0.

Updates the HTML representation with a style which is selected in accordance with the return value of a function.

Parameters

cond [callable] cond should take a scalar, and optional keyword arguments, and return a boolean.

value [str] Applied when cond returns true.

other [str] Applied when cond returns false.

subset [label, array-like, IndexSlice, optional] A valid 2d input to DataFrame.loc[<subset>], or, in the case of a 1d input or single key, to DataFrame.loc[:, <subset>] where the columns are prioritised, to limit data to before applying the function.

**kwargs [dict] Pass along to cond.

Returns

self [Styler]

See also:

Styler.applymap Apply a CSS-styling function elementwise.

Styler.apply Apply a CSS-styling function column-wise, row-wise, or table-wise.
Notes

This method is deprecated.

This method is a convenience wrapper for `Styler.applymap()`, which we recommend using instead.

The example:

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]])
>>> def cond(v, limit=4):
...     return v > 1 and v != limit
>>> df.style.where(cond, value='color:green;', other='color:red;')
```

should be refactored to:

```python
>>> def style_func(v, value, other, limit=4):
...     cond = v > 1 and v != limit
...     return value if cond else other
>>> df.style.applymap(style_func, value='color:green;', other='color:red;')
```

3.12.2 Styler properties

- `Styler.env`
- `Styler.template_html`
- `Styler.template_html_style`
- `Styler.template_html_table`
- `Styler.template_latex`
- `Styler.loader`

---

pandas.io.formats.style.Styler.env

Styler.env = <jinja2.environment.Environment object>

pandas.io.formats.style.Styler.template_html

Styler.template_html = <Template 'html.tpl'>
3.12.3 Style application

- **Styler.apply**(func[, axis, subset]) Apply a CSS-styling function column-wise, row-wise, or table-wise.
- **Styler.applymap**(func[, subset]) Apply a CSS-styling function elementwise.
- **Styler.apply_index**(func[, axis, level]) Apply a CSS-styling function to the index or column headers, level-wise.
- **Styler.applymap_index**(func[, axis, level]) Apply a CSS-styling function to the index or column headers, elementwise.
- **Styler.format**(formatter, subset, na_rep, ...) Format the text display value of cells.
- **Styler.format_index**(formatter, axis, ...) Format the text display value of index labels or column headers.
- **Styler.hide**(subset, axis, level, names) Hide the entire index / column headers, or specific rows / columns from display.
- **Styler.set_td_classes**(classes) Set the DataFrame of strings added to the class attribute of <td> HTML elements.
- **Styler.set_table_styles**(table_styles, ...) Set the table styles included within the <style> HTML element.
- **Styler.set_table_attributes**(attributes) Set the table attributes added to the <table> HTML element.
- **Styler.set_tooltips**(ttips[, props, css_class]) Set the DataFrame of strings on Styler generating hover tooltips.
- **Styler.set_caption**(caption) Set the text added to a <caption> HTML element.
- **Styler.set_sticky**(axis, pixel_size, levels) Add CSS to permanently display the index or column headers in a scrolling frame.
- **Styler.set_properties**(subset) Set defined CSS-properties to each <td> HTML element within the given subset.
- **Styler.set_uuid**(uuid) Set the uuid applied to id attributes of HTML elements.
- **Styler.clear**() Reset the Styler, removing any previously applied styles.
- **Styler.pipe**(func, *args, **kwargs) Apply func(self, *args, **kwargs), and return the result.
3.12.4 Builtin styles

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Styler.highlight_null</td>
<td>Highlight missing values with a style.</td>
</tr>
<tr>
<td>Styler.highlight_max</td>
<td>Highlight the maximum with a style.</td>
</tr>
<tr>
<td>Styler.highlight_min</td>
<td>Highlight the minimum with a style.</td>
</tr>
<tr>
<td>Styler.highlight_between</td>
<td>Highlight a defined range with a style.</td>
</tr>
<tr>
<td>Styler.highlight_quantile</td>
<td>Highlight values defined by a quantile with a style.</td>
</tr>
<tr>
<td>Styler.background_gradient</td>
<td>Color the background in a gradient style.</td>
</tr>
<tr>
<td>Styler.text_gradient</td>
<td>Color the text in a gradient style.</td>
</tr>
<tr>
<td>Styler.bar</td>
<td>Draw bar chart in the cell backgrounds.</td>
</tr>
</tbody>
</table>

3.12.5 Style export and import

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Styler.to_html</td>
<td>Write Styler to a file, buffer or string in HTML-CSS format.</td>
</tr>
<tr>
<td>Styler.to_latex</td>
<td>Write Styler to a file, buffer or string in LaTeX format.</td>
</tr>
<tr>
<td>Styler.to_excel</td>
<td>Write Styler to an Excel sheet.</td>
</tr>
<tr>
<td>Styler.export</td>
<td>Export the styles applied to the current Styler.</td>
</tr>
<tr>
<td>Styler.use</td>
<td>Set the styles on the current Styler.</td>
</tr>
</tbody>
</table>

3.13 Plotting

The following functions are contained in the pandas.plotting module.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>andrews_curves</td>
<td>Generate a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.</td>
</tr>
<tr>
<td>autocorrelation_plot</td>
<td>Autocorrelation plot for time series.</td>
</tr>
<tr>
<td>bootstrap_plot</td>
<td>Bootstrap plot on mean, median and mid-range statistics.</td>
</tr>
<tr>
<td>boxplot</td>
<td>Make a box plot from DataFrame columns.</td>
</tr>
<tr>
<td>deregister_matplotlib_converters</td>
<td>Remove pandas formatters and converters.</td>
</tr>
<tr>
<td>lag_plot</td>
<td>Lag plot for time series.</td>
</tr>
<tr>
<td>parallel_coordinates</td>
<td>Parallel coordinates plotting.</td>
</tr>
<tr>
<td>plot_params</td>
<td>Stores pandas plotting options.</td>
</tr>
<tr>
<td>radviz</td>
<td>Plot a multidimensional dataset in 2D.</td>
</tr>
<tr>
<td>register_matplotlib_converters</td>
<td>Register pandas formatters and converters with matplotlib.</td>
</tr>
<tr>
<td>scatter_matrix</td>
<td>Draw a matrix of scatter plots.</td>
</tr>
<tr>
<td>table</td>
<td>Helper function to convert DataFrame and Series to matplotlib.table.</td>
</tr>
</tbody>
</table>
3.13.1 pandas.plotting.andrews_curves

pandas.plotting.andrews_curves(frame, class_column, ax=None, samples=200, color=None, colormap=None, **kwargs)

Generate a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.

Andrews curves have the functional form:

\[ f(t) = \frac{x_1}{\sqrt{2}} + x_2 \sin(t) + x_3 \cos(t) + x_4 \sin(2t) + x_5 \cos(2t) + \ldots \]

Where x coefficients correspond to the values of each dimension and t is linearly spaced between -\pi and +\pi. Each row of frame then corresponds to a single curve.

Parameters

- **frame** [DataFrame] Data to be plotted, preferably normalized to (0.0, 1.0).
- **class_column** [Name of the column containing class names]
- **ax** [matplotlib axes object, default None]
- **samples** [Number of points to plot in each curve]
- **color** [list or tuple, optional] Colors to use for the different classes.
- **colormap** [str or matplotlib colormap object, default None] Colormap to select colors from.
  
  If string, load colormap with that name from matplotlib.
  
- **kwargs** Options to pass to matplotlib plotting method.

Returns

- class: *matplotlib.axis.Axes*

Examples

```python
>>> df = pd.read_csv(
...     'https://raw.github.com/pandas-dev/
...     'pandas/main/pandas/tests/io/data/csv/iris.csv'
... )
>>> pd.plotting.andrews_curves(df, 'Name')
<AxesSubplot:title={'center':'width'}>```

3.13.2 pandas.plotting.autocorrelation_plot

pandas.plotting.autocorrelation_plot(series, ax=None, **kwargs)

Autocorrelation plot for time series.

Parameters

- **series** [Time series]
- **ax** [Matplotlib axis object, optional]
- **kwargs** Options to pass to matplotlib plotting method.

Returns

- class: *matplotlib.axis.Axes*
3.13. Plotting

- Iris-setosa
- Iris-versicolor
- Iris-virginica
Examples

The horizontal lines in the plot correspond to 95% and 99% confidence bands.
The dashed line is 99% confidence band.

```python
>>> spacing = np.linspace(-9 * np.pi, 9 * np.pi, num=1000)
>>> s = pd.Series(0.7 * np.random.rand(1000) + 0.3 * np.sin(spacing))
>>> pd.plotting.autocorrelation_plot(s)
<AxesSubplot:title={'center':'width'}, xlabel='Lag', ylabel='Autocorrelation'>
```

3.13.3 pandas.plotting.bootstrap_plot

`pandas.plotting.bootstrap_plot(series, fig=None, size=50, samples=500, **kwds)`

Bootstrap plot on mean, median and mid-range statistics.

The bootstrap plot is used to estimate the uncertainty of a statistic by relaying on random sampling with replacement [1]. This function will generate bootstrapping plots for mean, median and mid-range statistics for the given number of samples of the given size.

**Parameters**

- `series` [pandas.Series] Series from where to get the samplings for the bootstrapping.
**fig** [matplotlib.figure.Figure, default None] If given, it will use the fig reference for plotting instead of creating a new one with default parameters.

**size** [int, default 50] Number of data points to consider during each sampling. It must be less than or equal to the length of the series.

**samples** [int, default 500] Number of times the bootstrap procedure is performed.

**kws** Options to pass to matplotlib plotting method.

**Returns**

matplotlib.figure.Figure Matplotlib figure.

**See also:**

DataFrame.plot Basic plotting for DataFrame objects.

Series.plot Basic plotting for Series objects.

**Examples**

This example draws a basic bootstrap plot for a Series.

```python
>>> s = pd.Series(np.random.uniform(size=100))
>>> pd.plotting.bootstrap_plot(s)
<Figure size 640x480 with 6 Axes>
```
### 3.13.4 pandas.plotting.boxplot

```
pandas.plotting.boxplot(data, column=None, by=None, ax=None, fontsize=None, rot=0, grid=True,
figsize=None, layout=None, return_type=None, **kwargs)
```

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. By default, they extend no more than $1.5 \times IQR$ ($IQR = Q3 - Q1$) from the edges of the box, ending at the farthest data point within that interval. Outliers are plotted as separate dots.

For further details see Wikipedia’s entry for boxplot.

**Parameters**

- **column** [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.
- **by** [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in `by`.
- **ax** [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.
- **fontsize** [float or str] Tick label font size in points or as a string (e.g., `large`).
- **rot** [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.
- **grid** [bool, default True] Setting this to True will show the grid.
- **figsize** [A tuple (width, height) in inches] The size of the figure to create in matplotlib.
- **layout** [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.
- **return_type** [{‘axes’, ‘dict’, ‘both’} or None, default ‘axes’] The kind of object to return. The default is `axes`.
  - ‘axes’ returns the matplotlib axes the boxplot is drawn on.
  - ‘dict’ returns a dictionary whose values are the matplotlib Lines of the boxplot.
  - ‘both’ returns a namedtuple with the axes and dict.
  - when grouping with by, a Series mapping columns to `return_type` is returned.
    If `return_type` is `None`, a NumPy array of axes with the same shape as `layout` is returned.
- **kwargs** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

**Returns**

- **result** See Notes.

See also:

- `Series.plot.hist` Make a histogram.
- `matplotlib.pyplot.boxplot` Matplotlib equivalent plot.
Notes

The return type depends on the `return_type` parameter:

- 'axes': object of class matplotlib.axes.Axes
- 'dict': dict of matplotlib.lines.Line2D objects
- 'both': a namedtuple with structure (ax, lines)

For data grouped with `by`, return a Series of the above or a numpy array:

- `Series`
- `array` (for `return_type = None`)

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```python
gnp.random.seed(1234)
df = pd.DataFrame(np.random.randn(10, 4),
... columns=['Col1', 'Col2', 'Col3', 'Col4'])
boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`.

3.13. Plotting 2685
For instance:

```python
>>> df = pd.DataFrame(np.random.randn(10, 2),
                    columns=['Col1', 'Col2'])
                      'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

![Boxplot grouped by X](image)

A list of strings (i.e. ['X', 'Y']) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

```python
>>> df = pd.DataFrame(np.random.randn(10, 3),
                    columns=['Col1', 'Col2', 'Col3'])
                      'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
                      'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

The layout of boxplot can be adjusted giving a tuple to `layout`:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
                        layout=(2, 1))
```
Additional formatting can be done to the boxplot, like suppressing the grid (grid=False), rotating the labels in the x-axis (i.e. rot=45) or changing the fontsize (i.e. fontsize=15):

```python
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

The parameter `return_type` can be used to select the type of element returned by `boxplot`. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with by, a Series mapping columns to `return_type` is returned:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
                         return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is `None`, a NumPy array of axes with the same shape as `layout` is returned:

```python
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
                         return_type=None)
```

(continues on next page)
3.13.5 pandas.plotting.deregister_matplotlib_converters

pandas.plotting.deregister_matplotlib_converters()

Remove pandas formatters and converters.

Removes the custom converters added by register(). This attempts to set the state of the registry back to the state before pandas registered its own units. Converters for pandas’ own types like Timestamp and Period are removed completely. Converters for types pandas overwrites, like datetime.datetime, are restored to their original value.

See also:

- register_matplotlib_converters  Register pandas formatters and converters with matplotlib.

3.13.6 pandas.plotting.lag_plot

pandas.plotting.lag_plot(series, lag=1, ax=None, **kwds)

Lag plot for time series.

**Parameters**

- series [Time series]
- lag [lag of the scatter plot, default 1]
- ax [Matplotlib axis object, optional]
- **kwds** Matplotlib scatter method keyword arguments.

**Returns**

- class:matplotlib.axis.Axes

**Examples**

Lag plots are most commonly used to look for patterns in time series data.

Given the following time series

```python
>>> np.random.seed(5)
>>> x = np.cumsum(np.random.normal(loc=1, scale=5, size=50))
>>> s = pd.Series(x)
>>> s.plot()
<AxesSubplot:xlabel='Midrange'>
```

A lag plot with lag=1 returns

```python
>>> pd.plotting.lag_plot(s, lag=1)
<AxesSubplot:xlabel='y(t)', ylabel='y(t + 1)'>
```
3.13.7 pandas.plotting.parallel_coordinates

```python
pandas.plotting.parallel_coordinates(frame, class_column, cols=None, ax=None, color=None,
    use_columns=False, xticks=None, colormap=None, axvlines=True,
    axvlines_kwds=None, sort_labels=False, **kwargs)
```

Parallel coordinates plotting.

**Parameters**

- **frame** [DataFrame]ibraltar.com/pandas-dev/
-
- **class_column** [str] Column name containing class names.
- **cols** [list, optional] A list of column names to use.
- **ax** [matplotlib.axis, optional] Matplotlib axis object.
- **color** [list or tuple, optional] Colors to use for the different classes.
- **use_columns** [bool, optional] If true, columns will be used as xticks.
- **xticks** [list or tuple, optional] A list of values to use for xticks.
- **colormap** [str or matplotlib colormap, default None] Colormap to use for line colors.
- **axvlines** [bool, optional] If true, vertical lines will be added at each xtick.
- **axvlines_kwds** [keywords, optional] Options to be passed to axvline method for vertical lines.
- **sort_labels** [bool, default False] Sort class_column labels, useful when assigning colors.

**kwargs** Options to pass to matplotlib plotting method.

**Returns**

- class: `matplotlib.axis.Axes`

**Examples**

```python
>>> df = pd.read_csv(
...     'https://raw.github.com/pandas-dev/
...     pandas/main/pandas/tests/io/data/csv/iris.csv'
... )
>>> pd.plotting.parallel_coordinates(
...     df, 'Name', color=('#566670', '#4ECDC4', '#C7F464')
... )
```

3.13.8 pandas.plotting.plot_params

```python
pandas.plotting.plot_params = {'xaxis_compat': False}
```

Stores pandas plotting options.

Allows for parameter aliasing so you can just use parameter names that are the same as the plot function parameters, but is stored in a canonical format that makes it easy to breakdown into groups later.
3.13.9 pandas.plotting.radviz

pandas.plotting.radviz(frame, class_column, ax=None, color=None, colormap=None, **kwds)

Plot a multidimensional dataset in 2D.

Each Series in the DataFrame is represented as a evenly distributed slice on a circle. Each data point is rendered in the circle according to the value on each Series. Highly correlated Series in the DataFrame are placed closer on the unit circle.

RadViz allow to project a N-dimensional data set into a 2D space where the influence of each dimension can be interpreted as a balance between the influence of all dimensions.

More info available at the original article describing RadViz.

Parameters

- frame [DataFrame] Object holding the data.
- class_column [str] Column name containing the name of the data point category.
- ax [matplotlib.axes.Axes, optional] A plot instance to which to add the information.
- color [list[set] or tuple[str], optional] Assign a color to each category. Example: ['blue', 'green'].
- colormap [str or matplotlib.colors.Colormap, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.
- **kwds Options to pass to matplotlib scatter plotting method.

Returns

class: matplotlib.axes.Axes

See also:

plotting.andrews_curves Plot clustering visualization.

Examples

```python
gp - df = pd.DataFrame(
... {  
... 'SepalLength': [6.5, 7.7, 5.1, 5.8, 7.6, 5.0, 5.4, 4.6, 6.7, 4.6],  
... 'SepalWidth': [3.0, 3.8, 3.8, 2.7, 3.0, 2.3, 3.0, 3.2, 3.3, 3.6],  
... 'PetalLength': [5.5, 6.7, 1.9, 5.1, 6.6, 3.3, 4.5, 1.4, 5.7, 1.0],  
... 'PetalWidth': [1.8, 2.2, 0.4, 1.9, 2.1, 1.0, 1.5, 0.2, 2.1, 0.2],  
... 'Category': ['virginica',  
... 'virginica',  
... 'setosa',  
... 'virginica',  
... 'virginica',  
... 'versicolor',  
... 'versicolor',  
... 'setosa',  
... 'virginica',  
... 'setosa']  
... })
```
>>> pd.plotting.radviz(df, 'Category')
<AxesSubplot:xlabel='y(t)', ylabel='y(t + 1)'>

3.13.10 pandas.plotting.register_matplotlib_converters

pandas.plotting.register_matplotlib_converters()

Register pandas formatters and converters with matplotlib.

This function modifies the global matplotlib.units.registry dictionary. pandas adds custom converters for
  • pd.Timestamp
  • pd.Period
  • np.datetime64
  • datetime.datetime
  • datetime.date
  • datetime.time

See also:

deregister_matplotlib_converters Remove pandas formatters and converters.
3.13.11 pandas.plotting.scatter_matrix

pandas.plotting.scatter_matrix(frame, alpha=0.5, figsize=None, ax=None, grid=False, diagonal='hist', marker='.', density_kwds=None, hist_kwds=None, range_padding=0.05, **kwargs)

Draw a matrix of scatter plots.

Parameters

frame [DataFrame] Amount of transparency applied.
alpha [float, optional] A tuple (width, height) in inches.
figsize [(float,float), optional] A Matplotlib axis object, optional
ax [bool, optional] Setting this to True will show the grid.
diagonal [{'hist', 'kde'}] Pick between ‘kde’ and ‘hist’ for either Kernel Density Estimation or Histogram plot in the diagonal.
marker [str, optional] Matplotlib marker type, default ‘.’.
density_kwds [keywords] Keyword arguments to be passed to kernel density estimate plot.
hist_kwds [keywords] Keyword arguments to be passed to hist function.
range_padding [float, default 0.05] Relative extension of axis range in x and y with respect to (x_max - x_min) or (y_max - y_min).

**kwargs [keywords] Keyword arguments to be passed to scatter function.

Returns

numpy.ndarray A matrix of scatter plots.

Examples

```python
>>> df = pd.DataFrame(np.random.randn(1000, 4), columns=['A', 'B', 'C', 'D'])
>>> pd.plotting.scatter_matrix(df, alpha=0.2)
array([[<AxesSubplot:xlabel='A', ylabel='A'>, ...
```

3.13. Plotting 2697
3.13.12 pandas.plotting.table

pandas.plotting[table](ax, data, rowLabels=None, colLabels=None, **kwargs)

Helper function to convert DataFrame and Series to matplotlib.table.

Parameters

ax [Matplotlib axes object]
data [DataFrame or Series] Data for table contents.

**kwargs Keyword arguments to be passed to matplotlib.table.table. If rowLabels or colLabels is not specified, data index or column name will be used.

Returns

matplotlib table object

3.14 General utility functions

3.14.1 Working with options

| describe_option(pat[, _print_desc]) | Prints the description for one or more registered options. |
| reset_option(pat) | Reset one or more options to their default value. |
| get_option(pat) | Retrieves the value of the specified option. |
| set_option(pat, value) | Sets the value of the specified option. |
| option_context(*args) | Context manager to temporarily set options in the with statement context. |

pandas.describe_option

pandas.describe_option(pat[, _print_desc=False] = <pandas._config.config.CallableDynamicDoc object>)

Prints the description for one or more registered options.

Call with no arguments to get a listing for all registered options.

Available options:

- compute.[use_bottleneck, use_numba, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_dir_items, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, min_rows, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- io.excel.ods.[reader, writer]
- io.excel.xls.[reader, writer]
- io.excel.xlsb.[reader]
- io.excel.xlsm.[reader, writer]
- io.excel.xlsx.[reader, writer]
Parameters

- **pat** [str] Regexp pattern. All matching keys will have their description displayed.
- **_print_desc** [bool, default True] If True (default) the description(s) will be printed to stdout. Otherwise, the description(s) will be returned as a unicode string (for testing).

Returns

None by default, the description(s) as a unicode string if _print_desc is False

Notes

Please reference the *User Guide* for more information.

The available options with its descriptions:

- **compute.use_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False, True [default: True] [currently: True]
- **compute.use_numba** [bool] Use the numba engine option for select operations if it is installed, the default is False Valid values: False, True [default: False] [currently: False]
- **compute.use_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False, True [default: True] [currently: True]
- **display.chop_threshold** [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]
- **display.colheader_justify** [‘left’/’right’] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]
- **display.column_space** No description available. [default: 12] [currently: 12]
- **display.date_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]
- **display.date_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]
- **display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: utf-8] [currently: utf-8]
- **display.expand_frame_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, max_columns is still respected, but the output will wrap-around across multiple “pages” if its width exceeds display.width. [default: True] [currently: True]
- **display.float_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]
- **display.html.border** [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1]
display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]
display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]
display.large_repr ['truncate’/’info’] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]
display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: False] [currently: False]
display.latex.longtable :bool This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]
display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]
display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]
display.latex.multirow [bool] This specifies if the to_latex method of a Dataframe uses multirow to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]
display.latex.repr [boolean] Whether to produce a latex DataFramerepresentation for jupyter environments that support it. (default: False) [default: False] [currently: False]
display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a Categorical or a Series of dtype “category”. [default: 8] [currently: 8]
display.max_columns [int] If max_cols is exceeded, switch to truncate view. Depending on large_repr, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]
display.max_colwidth [int or None] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “…” placeholder is embedded in the output. A ‘None’ value means unlimited. [default: 50] [currently: 50]
display.max_dir_items [int] The number of items that will be added to dir(...). ‘None’ value means unlimited.

Because dir is cached, changing this option will not immediately affect already existing dataframes until a column is deleted or added.

This is for instance used to suggest columns from a dataframe to tab completion. [default: 100] [currently: 100]
display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]
display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]
display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on large_repr, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 60]
display.max_seq_items [int or None] When pretty-printing a long sequence, no more then max_seq_items will be printed. If items are omitted, they will be denoted by the addition of “…” to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]
display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,’deep’ [default: True] [currently: True]
display.min_rows [int] The numbers of rows to show in a truncated view (when max_rows is exceeded). Ignored when max_rows is set to None or 0. When set to None, follows the value of max_rows. [default: 10] [currently: 10]
display.multi_sparse [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]
display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]
display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]
display.precision [int] Floating point output precision in terms of number of places after the decimal, for regular formatting as well as scientific notation. Similar to precision in numpy.set_printoptions(). [default: 6] [currently: 6]
display.show_dimensions [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]
display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect the performance (default: False) [currently: False]
display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect the performance (default: False) [currently: False]
display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]
io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]
mode.data_manager [string] Internal data manager type; can be “block” or “array”. Defaults to “block”, unless overridden by the ‘PANDAS_DATA_MANAGER’ environment variable (needs to be set before pandas is
imported). [default: block] [currently: block]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.string_storage [string] The default storage for StringDtype. [default: python] [currently: python]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use use_inf_as_na instead. [default: False] [currently: False] (Deprecated, use mode.use_inf_as_na instead.)

plotting.backend [str] The plotting backend to use. The default value is “matplotlib”, the backend provided with pandas. Other backends can be specified by providing the name of the module that implements the backend. [default: matplotlib] [currently: matplotlib]

plotting.matplotlib.register_converters [bool or ‘auto’.] Whether to register converters with matplotlib’s units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: auto] [currently: auto]

styler.format.decimal [str] The character representation for the decimal separator for floats and complex. [default: .] [currently: .]

styler.format.escape [str, optional] Whether to escape certain characters according to the given context: html or latex. [default: None] [currently: None]

styler.format.formatter [str, callable, dict, optional] A formatter object to be used as default within Styler. [default: None] [currently: None]

styler.format.na_rep [str, optional] The string representation for values identified as missing. [default: None] [currently: None]

styler.format.precision [int] The precision for floats and complex numbers. [default: 6] [currently: 6]

styler.format.thousands [str, optional] The character representation for thousands separator for floats, int and complex. [default: None] [currently: None]

styler.html.mathjax [bool] If False will render special CSS classes to table attributes that indicate Mathjax will not be used in Jupyter Notebook. [default: True] [currently: True]

styler.latex.environment [str] The environment to replace \begin{table}. If “longtable” is used results in a specific longtable environment format. [default: None] [currently: None]

styler.latex.hrules [bool] Whether to add horizontal rules on top and bottom and below the headers. [default: False] [currently: False]

styler.latex.multicol_align ["{r","c","l","naive-l","naive-r"}]

The specifier for horizontal alignment of sparsified LaTeX multicolumns. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. ‘|r’ will draw a rule on the left side of right aligned merged cells. [default: r] [currently: r]

styler.latex.multirow_align ["{c","t","b"}]
The specifier for vertical alignment of sparsified LaTeX multirows. [default: c] [currently: c]

styler.render.encoding [str] The encoding used for output HTML and LaTeX files. [default: utf-8] [currently: utf-8]

styler.render.max_columns [int, optional] The maximum number of columns that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

styler.render.max_elements [int] The maximum number of data-cell (<td>) elements that will be rendered before trimming will occur over columns, rows or both if needed. [default: 262144] [currently: 262144]

styler.render.max_rows [int, optional] The maximum number of rows that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

styler.render.repr [str] Determine which output to use in Jupyter Notebook in {“html”, “latex”}. [default: html] [currently: html]

styler.sparse.columns [bool] Whether to sparsify the display of hierarchical columns. Setting to False will display each explicit level element in a hierarchical key for each column. [default: True] [currently: True]

styler.sparse.index [bool] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. [default: True] [currently: True]
**pandas.reset_option**

`pandas.reset_option(pat) = <pandas._config.config.CallableDynamicDoc object>`

Reset one or more options to their default value.

Pass “all” as argument to reset all options.

Available options:

- `compute.[use_bottleneck, use_numba, use_numexpr]`
- `display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]`
- `display.html.[border, table_schema, use_mathjax]`
- `display.[large_repr]`
- `display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]`
- `display.[max_categories, max_columns, max_colwidth, max_dir_items, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, min_rows, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]`
- `display.unicode.[ambiguous_as_wide, east_asian_width]`
- `display.[width]`
- `io.excel.ods.[reader, writer]`
- `io.excel.xls.[reader, writer]`
- `io.excel.xlsb.[reader]`
- `io.excel.xlsm.[reader, writer]`
- `io.excel.xlsx.[reader, writer]`
- `io.hdf.[default_format, dropna_table]`
- `io.parquet.[engine]`
- `io.sql.[engine]`
- `mode.[chained_assignment, data_manager, sim_interactive, string_storage, use_inf_as_na, use_inf_as_null]`
- `plotting.[backend]`
- `plotting.matplotlib.[register_converters]`
- `styler.format.[decimal, escape, formatter, na_rep, precision, thousands]`
- `styler.html.[mathjax]`
- `styler.latex.[environment, hrules, multicol_align, multirow_align]`
- `styler.render.[encoding, max_columns, max_elements, max_rows, repr]`
- `styler.sparse.[columns, index]`

**Parameters**

- `pat` [str/regex] If specified only options matching `prefix*` will be reset. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. `x.y.z.option_name`), your code may break in future versions if new options with similar names are introduced.

**Returns**

- None
Notes

Please reference the User Guide for more information.

The available options with its descriptions:

- **compute.use_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True. Valid values: False, True [default: True] [currently: True]
- **compute.use_numba** [bool] Use the numba engine option for select operations if it is installed, the default is False. Valid values: False, True [default: False] [currently: False]
- **compute.use_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True. Valid values: False, True [default: True] [currently: True]
- **display.chop_threshold** [float or None] If set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]
- **display.colheader_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]
- **display.column_space** No description available. [default: 12] [currently: 12]
- **display.date_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]
- **display.date_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]
- **display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: utf-8] [currently: utf-8]
- **display.expand_frame_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, max_columns is still respected, but the output will wrap-around across multiple “pages” if its width exceeds display.width. [default: True] [currently: True]
- **display.float_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]
- **display.html.border** [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1]
- **display.html.table_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. [default: False] [currently: False]
- **display.html.use_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. [default: True] [currently: True]
- **display.large_repr** ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]
- **display.latex.escape** [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False, True [default: True] [currently: True]
- **display.latex.longtable** [bool] This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False, True [default: False] [currently: True]
- **display.latex.multicolumn** [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: True] [currently: True]
- **display.latex.multicolumn_format** [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: l] [currently: l]
- **display.latex.multirow** [bool] This specifies if the to_latex method of a Dataframe uses multirow to pretty-print MultiIndex rows. Valid values: False, True [default: False] [currently: False]
- **display.latex.repr** [bool] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False, True [default: False] [currently: False]
- **display.max_categories** [int] This sets the maximum number of categories pandas should output when printing out a Categorical or a Series of dtype “category”. [default: 8] [currently: 8]
- **display.max_columns** [int] If max_cols is exceeded, switch to truncate view. Depending on large_repr, objects
are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth  [int or None] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “…” placeholder is embedded in the output. A ‘None’ value means unlimited. [default: 50] [currently: 50]

display.max_dir_items [int] The number of items that will be added to dir(...). ‘None’ value means unlimited. Because dir is cached, changing this option will not immediately affect already existing dataframes until a column is deleted or added.

This is for instance used to suggest columns from a dataframe to tab completion. [default: 100] [currently: 100]

display.max_info_columns  [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows  [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on large_repr, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 60]

display.max_seq_items [int or None] When pretty-printing a long sequence, no more then max_seq_items will be printed. If items are omitted, they will be denoted by the addition of “…” to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage  [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,’deep’ [default: True] [currently: True]

display.min_rows  [int] The numbers of rows to show in a truncated view (when max_rows is exceeded). Ignored when max_rows is set to None or 0. When set to None, follows the value of max_rows. [default: 10] [currently: 10]

display.multi_sparse  [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html  [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth  [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision  [int] Floating point output precision in terms of number of places after the decimal, for regular formatting as well as scientific notation. Similar to precision in numpy.set_printoptions(). [default: 6] [currently: 6]

display.show_dimensions  [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide  [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width  [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython
qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]
```

```
io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
```

```
```

```
```

```
mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]
```

```
mode.data_manager [string] Internal data manager type; can be “block” or “array”. Defaults to “block”, unless overridden by the ‘PANDAS_DATA_MANAGER’ environment variable (needs to be set before pandas is imported). [default: block] [currently: block]
```

```
mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]
```

```
mode.string_storage [string] The default storage for StringDtype. [default: python] [currently: python]
```

```
mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]
```

```
mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use use_inf_as_na instead. [default: False] [currently: False] (Deprecated, use mode.use_inf_as_na instead.)
```

```
plotting.backend [str] The plotting backend to use. The default value is “matplotlib”, the backend provided with pandas. Other backends can be specified by providing the name of the module that implements the backend. [default: matplotlib] [currently: matplotlib]
```

```
plotting.matplotlib.register_converters [bool or ‘auto’] Whether to register converters with matplotlib’s units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: auto] [currently: auto]
```

```
styler.format.decimal [str] The character representation for the decimal separator for floats and complex. [default: ] [currently: ]
```

```
styler.format.escape [str, optional] Whether to escape certain characters according to the given context; html or latex. [default: None] [currently: None]
```

```
styler.format.formatter [str, callable, dict, optional] A formatter object to be used as default within Styler.
```

```
styler.format.na_rep [str, optional] The string representation for values identified as missing. [default: None] [currently: None]
```

3.14. General utility functions 2707
pandas: powerful Python data analysis toolkit, Release 1.4.2

styler.format.precision [int] The precision for floats and complex numbers. [default: 6] [currently: 6]
styler.format.thousands [str, optional] The character representation for thousands separator for floats, int and complex. [default: None] [currently: None]
styler.html.mathjax [bool] If False will render special CSS classes to table attributes that indicate Mathjax will not be used in Jupyter Notebook. [default: True] [currently: True]
styler.latex.environment [str] The environment to replace \begin{table}. If “longtable” is used results in a specific longtable environment format. [default: None] [currently: None]
styler.latex.hrules [bool] Whether to add horizontal rules on top and bottom and below the headers. [default: False] [currently: False]
styler.latex.multicol_align [""r", "c", "l", "naive-l", "naive-r"]] The specifier for horizontal alignment of sparsified LaTeX multicolumns. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. “|r” will draw a rule on the left side of right aligned merged cells. [default: r] [currently: r]
styler.latex.multirow_align ["c", "t", "b"]] The specifier for vertical alignment of sparsified LaTeX multirows. [default: c] [currently: c]
styler.render.encoding [str] The encoding used for output HTML and LaTeX files. [default: utf-8] [currently: utf-8]
styler.render.max_columns [int, optional] The maximum number of columns that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]
styler.render.max_elements [int] The maximum number of data-cell (<td>) elements that will be rendered before trimming will occur over columns, rows or both if needed. [default: 262144] [currently: 262144]
styler.render.max_rows [int, optional] The maximum number of rows that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]
styler.render.repr [str] Determines which output to use in Jupyter Notebook in {“html”, “latex”}. [default: html] [currently: html]
styler.sparse.columns [bool] Whether to sparsify the display of hierarchical columns. Setting to False will display each explicit level element in a hierarchical key for each column. [default: True] [currently: True]
styler.sparse.index [bool] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. [default: True] [currently: True]

pandas.get_option

pandas.get_option(pat) = <pandas._config.config.CallableDynamicDoc object>

Retrieves the value of the specified option.

Available options:
- compute.[use_bottleneck, use_numba, use_numexpr]
- display.chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format
- display.html.border, table_schema, use_mathjax
- display.large_repr
- display.latex.escape, longtable, multicolumn, multicolumn_format, multirow, repr
- display.max_categories, max_columns, max_colwidth, max_dir_items, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, min_rows, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions
- display.unicode.ambiguous_as_wide, east_asian_width
- display.width
- io.excel.ods.reader, writer
- io.excel.xls.reader, writer
- io.excel.xlsb.reader
- io.excel.xlsxsm.reader, writer
- io.excel.xlsx.reader, writer
- io.hdf.default_format, dropna_table
- io.parquet.engine
- io.sql.engine
• mode.[chained_assignment, data_manager, sim_interactive, string_storage, use_inf_as_na, use_inf_as_null]
• plotting.[backend]
• plotting.matplotlib.[register_converters]
• styler.format.[decimal, escape, formatter, na_rep, precision, thousands]
• styler.html.[mathjax]
• styler.latex.[environment, hrules, multicol_align, multirow_align]
• styler.render.[encoding, max_columns, max_elements, max_rows, repr]
• styler.sparse.[columns, index]

Parameters

pat [str] Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

Returns

result [the value of the option]

Raises

OptionError [if no such option exists]

Notes

Please reference the User Guide for more information.

The available options with its descriptions:
compute.use_bottleneck [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False, True [default: True] [currently: True]
compute.use_numba [bool] Use the numba engine option for select operations if it is installed, the default is False Valid values: False, True [default: False] [currently: False]
compute.use_numexpr [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False, True [default: True] [currently: True]
display.chop_threshold [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]
display.colheader_justify ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]
display.column_space No description available. [default: 12] [currently: 12]
display.date_dayfirst [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]
display.date_yearfirst [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]
display.encoding [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: utf-8] [currently: utf-8]
display.expand_frame_repr [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, max_columns is still respected, but the output will wrap-around across multiple "pages" if its width exceeds display.width. [default: True] [currently: True]
display.float_format [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]
display.html.border [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1]
display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]
display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]
display.large_repr [‘truncate’/’info’] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]
display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]
display.latex.longtable [bool] This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]
display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]
display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multi-columns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]
display.latex.repr [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]
display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a Categorical or a Series of dtype “category”. [default: 8] [currently: 8]
display.max_colwidth [int or None] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a ‘...’ placeholder is embedded in the output. A ‘None’ value means unlimited. [default: 50] [currently: 50]
display.max_dir_items [int] The number of items that will be added to dir(...). ‘None’ value means unlimited. Because dir is cached, changing this option will not immediately affect already existing dataframes until a column is deleted or added.

This is for instance used to suggest columns from a dataframe to tab completion. [default: 100] [currently: 100]
display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]
display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]
display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on large_repr, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]
display.max_seq_items [int or None] When pretty-printing a long sequence, no more then max_seq_items will be printed. If items are omitted, they will be denoted by the addition of ‘...’ to the resulting string. If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]
display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,’deep’ [default: True] [currently: True]
display.min_rows [int] The numbers of rows to show in a truncated view (when max_rows is exceeded). Ignored when max_rows is set to None or 0. When set to None, follows the value of max_rows. [default: 10]
display.multi_sparse [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]
display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]
display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]
display.precision [int] Floating point output precision in terms of number of places after the decimal, for regular formatting as well as scientific notation. Similar to precision in numpy.set_printoptions(). [default: 6] [currently: 6]
display.show_dimensions [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]
display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance [default: False] [currently: False]
display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance [default: False] [currently: False]
display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]
io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment. The default is warn [default: warn] [currently: warn]
mode.data_manager [string] Internal data manager type; can be “block” or “array”. Defaults to “block”, unless overridden by the ‘PANDAS_DATA_MANAGER’ environment variable (needs to be set before pandas is imported). [default: block] [currently: block]
mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False]
mode.string_storage  [string] The default storage for StringDtype. [default: python] [currently: python]
mode.use_inf_as_na  [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]
mode.use_inf_as_null  [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use use_inf_as_na instead. [default: False] [currently: False] (Deprecated, use mode.use_inf_as_na instead.)

plotting.backend  [str] The plotting backend to use. The default value is “matplotlib”, the backend provided with pandas. Other backends can be specified by providing the name of the module that implements the backend. [default: matlotlib] [currently: matlotlib]

plotting.matplotlib.register_converters  [bool or ‘auto’.] Whether to register converters with matplotlib’s units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: auto] [currently: auto]

styler.format.decimal  [str] The character representation for the decimal separator for floats and complex. [default: .] [currently: .]
styler.format.escape  [str, optional] Whether to escape certain characters according to the given context; html or latex. [default: None] [currently: None]
styler.format.formatter  [str, callable, dict, optional] A formatter object to be used as default within Styler. [default: None] [currently: None]
styler.format.na_rep  [str, optional] The string representation for values identified as missing. [default: None] [currently: None]
styler.format.precision  [int] The precision for floats and complex numbers. [default: 6] [currently: 6]
styler.format.thousands  [str, optional] The character representation for thousands separator for floats, int and complex. [default: None] [currently: None]

styler.html.mathjax  [bool] If False will render special CSS classes to table attributes that indicate Mathjax will not be used in Jupyter Notebook. [default: True] [currently: True]

styler.latex.environment  [str] The environment to replace \begin{table}. If “longtable” is used results in a specific longtable environment format. [default: None] [currently: None]

styler.latex.hrules  [bool] Whether to add horizontal rules on top and bottom and below the headers. [default: False] [currently: False]

styler.latex.multicol_align  [[“r”, “c”, “l”, “naive-l”, “naive-r”]] The specifier for horizontal alignment of sparsified LaTeX multicolumns. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. “|r” will draw a rule on the left side of right aligned merged cells. [default: r] [currently: r]

styler.latex.multirow_align  [[“c”, “t”, “b”]] The specifier for vertical alignment of sparsified LaTeX multirows. [default: c] [currently: c]

styler.render.encoding  [str] The encoding used for output HTML and LaTeX files. [default: utf-8] [currently: utf-8]

styler.render.max_columns  [int, optional] The maximum number of columns that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

styler.render.max_elements  [int] The maximum number of data-cell (<td>) elements that will be rendered before trimming will occur over columns, rows or both if needed. [default: 262144] [currently: 262144]

styler.render.max_rows  [int, optional] The maximum number of rows that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

styler.render.repr  [str] Determine which output to use in Jupyter Notebook in {“html”, “latex”}. [default: html] [currently: html]

stylersparse.columns  [bool] Whether to sparsify the display of hierarchical columns. Setting to False will display each explicit level element in a hierarchical key for each column. [default: True] [currently: True]

stylersparse.index  [bool] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. [default: True] [currently: True]
pandas.set_option

pandas.set_option(pat, value) = <pandas._config.config.CallableDynamicDoc object>

Sets the value of the specified option.

Available options:

- compute.[use_bottleneck, use_numba, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_dir_items, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, min_rows, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- io.excel.ods.[reader, writer]
- io.excel.xls.[reader, writer]
- io.excel.xlsb.[reader]
- io.excel.xlsm.[reader, writer]
- io.excel.xlsx.[reader, writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- io.sql.[engine]
- mode.[chained_assignment, data_manager, sim_interactive, string_storage, use_inf_as_na, use_inf_as_null]
- plotting.[backend]
- plotting.matplotlib.[register_converters]
- styler.format.[decimal, escape, formatter, na_rep, precision, thousands]
- styler.html.[mathjax]
- styler.latex.[environment, hrules, multicolumn, multirow]
- styler.render.[encoding, max_columns, max_elements, max_rows, repr]
- styler.sparse.[columns, index]

Parameters

- pat [str] Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

- value [object] New value of option.

Returns

- None

Raises

- OptionError if no such option exists
Notes

Please reference the *User Guide* for more information.

The available options with its descriptions:

**compute.use_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True. Valid values: False, True [default: True] [currently: True]

**compute.use_numba** [bool] Use the numba engine option for select operations if it is installed, the default is False. Valid values: False, True [default: False] [currently: False]

**compute.use_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True. Valid values: False, True [default: True] [currently: True]

**display.chop_threshold** [float or None] If set to a float value, all float values smaller than the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

**display.colheader_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

**display.column_space** No description available. [default: 12] [currently: 12]

**display.date_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005. [default: False] [currently: False]

**display.date_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20. [default: False] [currently: False]

**display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: utf-8] [currently: utf-8]

**display.expand_frame_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, max_columns is still respected, but the output will wrap-around across multiple “pages” if its width exceeds display.width. [default: True] [currently: True]

**display.float_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

**display.html.border** [int] A border=value attribute is inserted in the <table> tag for the DataFrame HTML repr. [default: 1] [currently: 1]

**display.html.table_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. [default: False] [currently: False]

**display.html.use_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. [default: True] [currently: True]

**display.large_repr** ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

**display.latex.escape** [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False, True [default: True] [currently: True]

**display.latex.longtable** [boolean] This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False, True [default: True] [currently: True]

**display.latex.multicolumn** [boolean] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: True] [currently: True]

**display.latex.multicolumn_format** [boolean] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: l] [currently: l]

**display.latex.multirow** [boolean] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False, True [default: False] [currently: False]

**display.latex.repr** [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. [default: False] [currently: False]

**display.max_categories** [int] This sets the maximum number of categories pandas should output when printing out a Categorical or a Series of dtype “category”. [default: 8] [currently: 8]

**display.max_columns** [int] If max_cols is exceeded, switch to truncate view. Depending on large_repr, objects
are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth [int or None] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “…” placeholder is embedded in the output. A ‘None’ value means unlimited. [default: 50] [currently: 50]

display.max_dir_items [int] The number of items that will be added to dir(...). ‘None’ value means unlimited. Because dir is cached, changing this option will not immediately affect already existing dataframes until a column is deleted or added.

This is for instance used to suggest columns from a dataframe to tab completion. [default: 100] [currently: 100]

display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on large_repr, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and large_repr equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 60]

display.max_seq_items [int or None] When pretty-printing a long sequence, no more then max_seq_items will be printed. If items are omitted, they will be denoted by the addition of “…” to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True, False, ‘deep’ [default: True] [currently: True]

display.min_rows [int] The numbers of rows to show in a truncated view (when max_rows is exceeded). Ignored when max_rows is set to None or 0. When set to None, follows the value of max_rows. [default: 10] [currently: 10]

display.multi_sparse [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision [int] Floating point output precision in terms of number of places after the decimal, for regular formatting as well as scientific notation. Similar to precision in numpy.set_printoptions(). [default: 6] [currently: 6]

display.show_dimensions [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython
qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width.
[default: 80] [currently: 80]

io.hdf.default_format [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]
io.hdf.droprna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]

mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment. The default is warn [default: warn] [currently: warn]

mode.data_manager [string] Internal data manager type; can be “block” or “array”. Default to “block”, unless overridden by the ‘PANDAS_DATA_MANAGER’ environment variable (needs to be set before pandas is imported). [default: block] [currently: block]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.string_storage [string] The default storage for StringDtype. [default: python] [currently: python]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way). False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] Use use_inf_as_null instead. [default: False] [currently: False] (Deprecated, use mode.use_inf_as_na instead.)

plotting.backend [str] The plotting backend to use. The default value is “matplotlib”, the backend provided with pandas. Other backends can be specified by providing the name of the module that implements the backend. [default: matplotlib] [currently: matplotlib]

plotting.matplotlib.register_converters [bool or ‘auto’] Whether to register converters with matplotlib’s units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overrode. [default: auto] [currently: auto]

styler.format.decimal [str] The character representation for the decimal separator for floats and complex. [default: .] [currently: .]

styler.format.escape [str, optional] Whether to escape certain characters according to the given context; html or latex. [default: None] [currently: None]

styler.format.formatter [str, callable, dict, optional] A formatter object to be used as default within Styler. [default: None] [currently: None]

styler.format.na_rep [str, optional] The string representation for values identified as missing. [default: None] [currently: None]
**pandas**: powerful Python data analysis toolkit, Release 1.4.2

**styler.format.precision** [int] The precision for floats and complex numbers. [default: 6] [currently: 6]

**styler.format.thousands** [str, optional] The character representation for thousands separator for floats, int and complex. [default: None] [currently: None]

**styler.html.mathjax** [bool] If False will render special CSS classes to table attributes that indicate Mathjax will not be used in Jupyter Notebook. [default: True] [currently: True]

**styler.latex.environment** [str] The environment to replace \begin{table}. If "longtable" is used results in a specific longtable environment format. [default: None] [currently: None]

**styler.latex.hrules** [bool] Whether to add horizontal rules on top and bottom and below the headers. [default: False] [currently: False]

**styler.latex.multicol_align** ["r", "c", "l", "naive-l", "naive-r"] The specifier for horizontal alignment of sparsified LaTeX multicolumns. Pipe decorators can also be added to non-naive values to draw vertical rules, e.g. "r|l" will draw a rule on the left side of right aligned merged cells. [default: r] [currently: r]

**styler.latex.multirow_align** ["c", "t", "b"] The specifier for vertical alignment of sparsified LaTeX multirows. [default: c] [currently: c]

**styler.render.encoding** [str] The encoding used for output HTML and LaTeX files. [default: utf-8] [currently: utf-8]

**styler.render.max_columns** [int, optional] The maximum number of columns that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

**styler.render.max_elements** [int] The maximum number of data-cell (<td>) elements that will be rendered before trimming will occur over columns, rows or both if needed. [default: 262144] [currently: 262144]

**styler.render.max_rows** [int, optional] The maximum number of rows that will be rendered. May still be reduced to satisfy max_elements, which takes precedence. [default: None] [currently: None]

**styler.render.repr** [str] Determine which output to use in Jupyter Notebook in {"html", "latex"}. [default: html] [currently: html]

**styler.sparse.columns** [bool] Whether to sparsify the display of hierarchical columns. Setting to False will display each explicit level element in a hierarchical key for each column. [default: True] [currently: True]

**styler.sparse.index** [bool] Whether to sparsify the display of a hierarchical index. Setting to False will display each explicit level element in a hierarchical key for each row. [default: True] [currently: True]

---

### pandas.option_context

**class** pandas.option_context(*args)

Context manager to temporarily set options in the with statement context.

You need to invoke as option_context(pat, val, [(pat, val), ...]).

**Examples**

```python
>>> with option_context('display.max_rows', 10, 'display.max_columns', 5):
...     pass
```

**Methods**

```python
__call__(func) Call self as a function.
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
pandas.option_context.__call__

option_context.__call__(func)
Call self as a function.
```

### 3.14.2 Testing functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>testing.assert_frame_equal(left, right[, ...])</code></td>
<td>Check that left and right DataFrame are equal.</td>
</tr>
<tr>
<td><code>testing.assert_series_equal(left, right[, ...])</code></td>
<td>Check that left and right Series are equal.</td>
</tr>
<tr>
<td><code>testing.assert_index_equal(left, right[, ...])</code></td>
<td>Check that left and right Index are equal.</td>
</tr>
<tr>
<td><code>testing.assert_extension_array_equal(left, right)</code></td>
<td>Check that left and right ExtensionArrays are equal.</td>
</tr>
</tbody>
</table>

**pandas.testing.assert_frame_equal**

```python
pandas.testing.assert_frame_equal(left, right, check_dtype=True,
check_index_type='equiv',
check_column_type='equiv', check_frame_type=True,
check_less_precise=False, check_names=True,
by_blocks=False, check_exact=False, check_datetimelike_compat=False,
check_like=False, check_freq=True, check_flags=True,
rtol=1e-05, atol=1e-08, obj='DataFrame')
```

Check that left and right DataFrame are equal.

This function is intended to compare two DataFrames and output any differences. It is mostly intended for use in unit tests. Additional parameters allow varying the strictness of the equality checks performed.

**Parameters**

- **left** [DataFrame] First DataFrame to compare.
- **right** [DataFrame] Second DataFrame to compare.
- **check_dtype** [bool, default True] Whether to check the DataFrame dtype is identical.
- **check_index_type** [bool or {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and inferred_type are identical.
- **check_column_type** [bool or {'equiv'}, default 'equiv'] Whether to check the columns class, dtype and inferred_type are identical. Is passed as the exact argument of `assert_index_equal()`.
- **check_frame_type** [bool, default True] Whether to check the DataFrame class is identical.
- **check_less_precise** [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare.

When comparing two numbers, if the first number has magnitude less than 1e-5, we compare the two numbers directly and check whether they are equivalent within the specified precision. Otherwise, we compare the ratio of the second number to the first number and check whether it is equivalent to 1 within the specified precision.

Deprecation since version 1.1.0: Use `rtol` and `atol` instead to define relative/absolute tolerance, respectively. Similar to `math.isclose()`.

2718 Chapter 3. API reference
check_names [bool, default True] Whether to check that the names attribute for both the index and column attributes of the DataFrame is identical.

by_blocks [bool, default False] Specify how to compare internal data. If False, compare by columns. If True, compare by blocks.

check_exact [bool, default False] Whether to compare number exactly.

check_datetimelike_compat [bool, default False] Compare datetime-like which is comparable ignoring dtype.

check_categorical [bool, default True] Whether to compare internal Categorical exactly.

check_like [bool, default False] If True, ignore the order of index & columns. Note: index labels must match their respective rows (same as in columns) - same labels must be with the same data.

check_freq [bool, default True] Whether to check the freq attribute on a DatetimeIndex or TimedeltaIndex.

New in version 1.1.0.

check_flags [bool, default True] Whether to check the flags attribute.

rtol [float, default 1e-5] Relative tolerance. Only used when check_exact is False.

New in version 1.1.0.

atol [float, default 1e-8] Absolute tolerance. Only used when check_exact is False.

New in version 1.1.0.

obj [str, default ‘DataFrame’] Specify object name being compared, internally used to show appropriate assertion message.

See also:

assert_series_equal Equivalent method for asserting Series equality.

DataFrame.equals Check DataFrame equality.

Examples

This example shows comparing two DataFrames that are equal but with columns of differing dtypes.

```python
>>> from pandas.testing import assert_frame_equal
>>> df1 = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
>>> df2 = pd.DataFrame({'a': [1, 2], 'b': [3.0, 4.0]})
```

df1 equals itself.

```python
>>> assert_frame_equal(df1, df1)
```

df1 differs from df2 as column ‘b’ is of a different type.

```python
>>> assert_frame_equal(df1, df2)
Traceback (most recent call last):
...
AssertionError: Attributes of DataFrame.iloc[:, 1] (column name="b") are different
```

Attribute “dtype” are different [left]: int64 [right]: float64

Ignore differing dtypes in columns with check_dtype.
>>> assert_frame_equal(df1, df2, check_dtype=False)

pandas.testing.assert_series_equal

pandas.testing.assert_series_equal(left, right, check_dtype=True, check_index_type='equiv',
check_series_type=True, check_less_precise=NoDefault.no_default,
check_names=True, check_exact=False,
check_datetimelike_compat=False, check_categorical=True,
check_category_order=True, check_freq=True, check_flags=True,
rtol=1e-05, atol=1e-08, obj='Series', *, check_index=True)

Check that left and right Series are equal.

Parameters

left [Series]
right [Series]
check_dtype [bool, default True] Whether to check the Series dtype is identical.
check_index_type [bool or {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and inferred_type are identical.
check_series_type [bool, default True] Whether to check the Series class is identical.
check_less_precise [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare.
When comparing two numbers, if the first number has magnitude less than 1e-5, we compare the two numbers directly and check whether they are equivalent within the specified precision. Otherwise, we compare the ratio of the second number to the first number and check whether it is equivalent to 1 within the specified precision.
Deprecated since version 1.1.0: Use rtol and atol instead to define relative/absolute tolerance, respectively. Similar to math.isclose().
check_names [bool, default True] Whether to check the Series and Index names attribute.
check_exact [bool, default False] Whether to compare number exactly.
check_datetimelike_compat [bool, default False] Compare datetime-like which is comparable ignoring dtype.
check_categorical [bool, default True] Whether to compare internal Categorical exactly.
check_category_order [bool, default True] Whether to compare category order of internal Categoricals.
New in version 1.0.2.
check_freq [bool, default True] Whether to check the freq attribute on a DatetimeIndex or TimedeltaIndex.
New in version 1.1.0.
check_flags [bool, default True] Whether to check the flags attribute.
New in version 1.2.0.
rtol [float, default 1e-5] Relative tolerance. Only used when check_exact is False.
New in version 1.1.0.
atol  [float, default 1e-8] Absolute tolerance. Only used when check_exact is False.
    New in version 1.1.0.

obj  [str, default ‘Series’] Specify object name being compared, internally used to show ap-
    propriate assertion message.

check_index  [bool, default True] Whether to check index equivalence. If False, then compare
    only values.
    New in version 1.3.0.

Examples

```python
>>> from pandas import testing as tm
>>> a = pd.Series([1, 2, 3, 4])
>>> b = pd.Series([1, 2, 3, 4])
>>> tm.assert_series_equal(a, b)
```

pandas.testing.assert_index_equal

```python
def assert_index_equal(left, right, exact='equiv', check_names=True,
                       check_less_precise=False, check_exact=True,
                       check_categorical=True, check_order=True, rtol=1e-5, atol=1e-8,
                       obj='Index')
```

Check that left and right Index are equal.

Parameters

- `left`  [Index]
- `right`  [Index]
- `exact`  [bool or {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and in-
    ferred_type are identical. If ‘equiv’, then RangeIndex can be substituted for Int64Index
    as well.
- `check_names`  [bool, default True] Whether to check the names attribute.
- `check_less_precise`  [bool or int, default False] Specify comparison precision. Only used
    when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are
    compared. If int, then specify the digits to compare.
    Deprecated since version 1.1.0: Use rtol and atol instead to define relative/absolute
    tolerance, respectively. Similar to math.isclose().
- `check_exact`  [bool, default True] Whether to compare number exactly.
- `check_categorical`  [bool, default True] Whether to compare internal Categorical exactly.
- `check_order`  [bool, default True] Whether to compare the order of index entries as well as
    their values. If True, both indexes must contain the same elements, in the same order. If
    False, both indexes must contain the same elements, but in any order.
    New in version 1.2.0.
- `rtol`  [float, default 1e-5] Relative tolerance. Only used when check_exact is False.
    New in version 1.1.0.
pandas: powerful Python data analysis toolkit, Release 1.4.2

atol [float, default 1e-8] Absolute tolerance. Only used when check_exact is False.

New in version 1.1.0.

obj [str, default ‘Index’] Specify object name being compared, internally used to show appropriate assertion message.

Examples

```python
>>> from pandas import testing as tm
>>> a = pd.Index([1, 2, 3])
>>> b = pd.Index([1, 2, 3])
>>> tm.assert_index_equal(a, b)
```

pandas.testing.assert_extension_array_equal

pandas.testing.assert_extension_array_equal(left, right, check_dtype=True, index_values=None, check_less_precise=False, check_exact=False, rtol=1e-05, atol=1e-08)

Check that left and right ExtensionArrays are equal.

Parameters

left, right [ExtensionArray] The two arrays to compare.

check_dtype [bool, default True] Whether to check if the ExtensionArray dtypes are identical.

index_values [numpy.ndarray, default None] Optional index (shared by both left and right), used in output.

check_less_precise [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare.

Deprecated since version 1.1.0: Use rtol and atol instead to define relative/absolute tolerance, respectively. Similar to math.isclose().

check_exact [bool, default False] Whether to compare number exactly.

rtol [float, default 1e-5] Relative tolerance. Only used when check_exact is False.

New in version 1.1.0.

atol [float, default 1e-8] Absolute tolerance. Only used when check_exact is False.

New in version 1.1.0.

Notes

Missing values are checked separately from valid values. A mask of missing values is computed for each and checked to match. The remaining all-valid values are cast to object dtype and checked.
### Examples

```python
>>> from pandas import testing as tm
>>> a = pd.Series([1, 2, 3, 4])
>>> b, c = a.array, a.array
>>> tm.assert_extension_array_equal(b, c)
```

## 3.14.3 Exceptions and warnings

<table>
<thead>
<tr>
<th>Error Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errors.AbstractMethodError</code></td>
<td>Raise this error instead of <code>NotImplementedError</code> for abstract methods while keeping compatibility with Python 2 and Python 3.</td>
</tr>
<tr>
<td><code>errors.AccessorRegistrationWarning</code></td>
<td>Warning for attribute conflicts in accessor registration.</td>
</tr>
<tr>
<td><code>errors.DtypeWarning</code></td>
<td>Warning raised when reading different dtypes in a column from a file.</td>
</tr>
<tr>
<td><code>errors.DuplicateLabelError</code></td>
<td>Error raised when an operation would introduce duplicate labels.</td>
</tr>
<tr>
<td><code>errors.EmptyDataError</code></td>
<td>Exception that is thrown in <code>pd.read_csv</code> (by both the C and Python engines) when empty data or header is encountered.</td>
</tr>
<tr>
<td><code>errors.InvalidIndexError</code></td>
<td>Exception raised when attempting to use an invalid index key.</td>
</tr>
<tr>
<td><code>errors.IntCastingNaNError</code></td>
<td>Raised when attempting an astype operation on an array with NaN to an integer dtype.</td>
</tr>
<tr>
<td><code>errors.MergeError</code></td>
<td>Error raised when problems arise during merging due to problems with input data.</td>
</tr>
<tr>
<td><code>errors.NullFrequencyError</code></td>
<td>Error raised when a null <code>freq</code> attribute is used in an operation that needs a non-null frequency, particularly <code>Date-</code> <code>TimeIndex.shift, TimedeltaIndex shift, PeriodIndex.shift</code>.</td>
</tr>
<tr>
<td><code>errors.OptionError</code></td>
<td>Exception for <code>pandas.options</code>, backwards compatible with <code>KeyError</code> checks.</td>
</tr>
<tr>
<td><code>errorsOutOfBoundsDatetime</code></td>
<td></td>
</tr>
<tr>
<td><code>errorsOutOfBoundsTimedelta</code></td>
<td></td>
</tr>
<tr>
<td><code>errors.ParserError</code></td>
<td>Exception that is raised by an error encountered in parsing file contents.</td>
</tr>
<tr>
<td><code>errors.ParserWarning</code></td>
<td>Warning raised when reading a file that doesn’t use the default ‘c’ parser.</td>
</tr>
<tr>
<td><code>errors.PerformanceWarning</code></td>
<td>Warning raised when there is a possible performance impact.</td>
</tr>
<tr>
<td><code>errors.UnsortedIndexError</code></td>
<td>Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted.</td>
</tr>
<tr>
<td><code>errors.UnsupportedFunctionCall</code></td>
<td>Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g.</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

pandas.errors(AbstractMethodError

exception pandas.errors.AbstractMethodError(class_instance, methodtype='method')
    Raise this error instead of NotImplementedError for abstract methods while keeping compatibility with Python 2 and Python 3.

pandas.errors.AccessorRegistrationWarning

exception pandas.errors.AccessorRegistrationWarning
    Warning for attribute conflicts in accessor registration.

pandas.errors(DtypeWarning

exception pandas.errors.DtypeWarning
    Warning raised when reading different dtypes in a column from a file.
    Raised for a dtype incompatibility. This can happen whenever read_csv or read_table encounter non-uniform dtypes in a column(s) of a given CSV file.

    See also:
    read_csv Read CSV (comma-separated) file into a DataFrame.
    read_table Read general delimited file into a DataFrame.

Notes

This warning is issued when dealing with larger files because the dtype checking happens per chunk read.

Despite the warning, the CSV file is read with mixed types in a single column which will be an object type. See the examples below to better understand this issue.

Examples

This example creates and reads a large CSV file with a column that contains int and str.

```python
>>> df = pd.DataFrame({
    'a': ([1] * 100000 + [X] * 100000 +
        ... [1] * 100000),
    ... 'b': [b] * 300000})
>>> df.to_csv('test.csv', index=False)
>>> df2 = pd.read_csv('test.csv')
    ... # DtypeWarning: Columns (0) have mixed types

Important to notice that df2 will contain both str and int for the same input, ‘1’.

```
One way to solve this issue is using the `dtype` parameter in the `read_csv` and `read_table` functions to explicit the conversion:

```python
>>> df2 = pd.read_csv('test.csv', sep=',', dtype={'a': str})
```

No warning was issued.

### pandas.errors.DuplicateLabelError

**exception** `pandas.errors.DuplicateLabelError`

Error raised when an operation would introduce duplicate labels.

New in version 1.2.0.

**Examples**

```python
>>> s = pd.Series([0, 1, 2], index=['a', 'b', 'c']).set_flags(
...    allows_duplicate_labels=False
...)
...)
>>> s.reindex(['a', 'a', 'b'])
Traceback (most recent call last):
  ...  
DuplicateLabelError: Index has duplicates.
  positions
table
a  [0, 1]
```

### pandas.errors.EmptyDataError

**exception** `pandas.errors.EmptyDataError`

Exception that is thrown in `pd.read_csv` (by both the C and Python engines) when empty data or header is encountered.

### pandas.errors.InvalidIndexError

**exception** `pandas.errors.InvalidIndexError`

Exception raised when attempting to use an invalid index key.

New in version 1.1.0.

### pandas.errors.IntCastingNaNError

**exception** `pandas.errors.IntCastingNaNError`

Raised when attempting an astype operation on an array with NaN to an integer dtype.
pandas.errors.MergeError

exception pandas.errors.MergeError
   Error raised when problems arise during merging due to problems with input data. Subclass of ValueError.

pandas.errors.NullFrequencyError

exception pandas.errors.NullFrequencyError
   Error raised when a null freq attribute is used in an operation that needs a non-null frequency, particularly DateTimeIndex.shift, TimedeltaIndex.shift, PeriodIndex.shift.

pandas.errors.NumbaUtilError

exception pandas.errors.NumbaUtilError
   Error raised for unsupported Numba engine routines.

pandas.errors.OptionError

exception pandas.errors.OptionError
   Exception for pandas.options, backwards compatible with KeyError checks.

pandas.errors.OutOfBoundsDatetime

exception pandas.errors.OutOfBoundsDatetime

pandas.errors.OutOfBoundsTimedelta

exception pandas.errors.OutOfBoundsTimedelta
   Raised when encountering a timedelta value that cannot be represented as a timedelta64[ns].

pandas.errors.ParserError

exception pandas.errors.ParserError
   Exception that is raised by an error encountered in parsing file contents.

   This is a generic error raised for errors encountered when functions like read_csv or read_html are parsing contents of a file.

   See also:

   read_csv  Read CSV (comma-separated) file into a DataFrame.
   read_html Read HTML table into a DataFrame.
pandas.errors.ParserWarning

exception pandas.errors.ParserWarning
Warning raised when reading a file that doesn’t use the default ‘c’ parser.

Raised by pd.read_csv and pd.read_table when it is necessary to change parsers, generally from the default ‘c’ parser to ‘python’.

It happens due to a lack of support or functionality for parsing a particular attribute of a CSV file with the requested engine.

Currently, ‘c’ unsupported options include the following parameters:
1. sep other than a single character (e.g. regex separators)
2. skipfooter higher than 0
3. sep=None with delim_whitespace=False

The warning can be avoided by adding engine='python' as a parameter in pd.read_csv and pd.read_table methods.

See also:
pd.read_csv  Read CSV (comma-separated) file into DataFrame.
pd.read_table  Read general delimited file into DataFrame.

Examples

Using a sep in pd.read_csv other than a single character:

```python
>>> import io
>>> csv = '''a;b;c
... 1;1,8
... 1;2,1'''
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]')
... # ParserWarning: Falling back to the 'python' engine...
```

Adding engine='python' to pd.read_csv removes the Warning:

```python
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]', engine='python')
```

pandas.errors.PerformanceWarning

exception pandas.errors.PerformanceWarning
Warning raised when there is a possible performance impact.

pandas.errors.UnsortedIndexError

exception pandas.errors.UnsortedIndexError
Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted. Subclass of KeyError.
### pandas.errors.UnsupportedFunctionCall

**exception** pandas.errors.UnsupportedFunctionCall

Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g. `np.cumsum(groupby_object)`.

#### 3.14.4 Data types related functionality

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api.types.union_categoricals(to_union[, ...])</code></td>
<td>Combine list-like of Categorical-like, unioning categories.</td>
</tr>
<tr>
<td><code>api.types.infer_dtype</code></td>
<td>Efficiently infer the type of a passed val, or list-like array of values.</td>
</tr>
<tr>
<td><code>api.types.pandas_dtype(dtype)</code></td>
<td>Convert input into a pandas only dtype object or a numpy dtype object.</td>
</tr>
</tbody>
</table>

**pandas.api.types.union_categoricals**

`pandas.api.types.union_categoricals(to_union, sort_categories=False, ignore_order=False)`

Combine list-like of Categorical-like, unioning categories.

All categories must have the same dtype.

**Parameters**

- **to_union** [list-like] Categorical, CategoricalIndex, or Series with dtype='category'.
- **sort_categories** [bool, default False] If true, resulting categories will be lexsorted, otherwise they will be ordered as they appear in the data.
- **ignore_order** [bool, default False] If true, the ordered attribute of the Categoricals will be ignored. Results in an unordered categorical.

**Returns**

- Categorical

**Raises**

**TypeError**

- all inputs do not have the same dtype
- all inputs do not have the same ordered property
- all inputs are ordered and their categories are not identical
- sort_categories=True and Categoricals are ordered

**ValueError** Empty list of categoricals passed
Notes

To learn more about categories, see link

Examples

```python
from pandas.api.types import union_categoricals

If you want to combine categoricals that do not necessarily have the same categories, `union_categoricals` will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```python
d = pd.Categorical(['b', 'c'])
e = pd.Categorical(['a', 'b'])
union_categoricals([d, e])
```

Categories (3, object): ['b', 'c', 'a']

By default, the resulting categories will be ordered as they appear in the `categories` of the data. If you want the categories to be lexsorted, use `sort_categories=True` argument.

```python
d = pd.Categorical(['b', 'c'], ordered=True)
e = pd.Categorical(['a', 'b'], ordered=True)
union_categoricals([d, e], sort_categories=True)
```

Categories (3, object): ['a', 'b', 'c']

`union_categoricals` also works with the case of combining two categoricals of the same categories and order information (e.g. what you could also `append` for).

```python
d = pd.Categorical(['a', 'b', 'c'], ordered=True)
e = pd.Categorical(['c', 'b', 'a'], ordered=True)
union_categoricals([d, e])
```

Categories (2, object): ['a' < 'b']

Raises `TypeError` because the categories are ordered and not identical.

```python
d = pd.Categorical(['a', 'b'], ordered=True)
e = pd.Categorical(['a', 'b', 'c'], ordered=True)
union_categoricals([d, e])
```

```
Traceback (most recent call last):
...
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```python
d = pd.Categorical(['a', 'b', 'c'], ordered=True)
e = pd.Categorical(['c', 'b', 'a'], ordered=True)
union_categoricals([d, e], ignore_order=True)
```

Categories (3, object): ['a', 'b', 'c']

`union_categoricals` also works with a `CategoricalIndex`, or `Series` containing categorical data, but note that the resulting array will always be a plain `Categorical`
>>> a = pd.Series(['b', 'c'], dtype='category')
>>> b = pd.Series(['a', 'b'], dtype='category')
>>> union_categoricals([a, b])
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']

**pandas.api.types.infer_dtype**

pandas.api.types.infer_dtype()

Efficiently infer the type of a passed val, or list-like array of values. Return a string describing the type.

**Parameters**

- **value** [scalar, list, ndarray, or pandas type]
- **skipna** [bool, default True] Ignore NaN values when inferring the type.

**Returns**

- **str** Describing the common type of the input data.

**Results can include:**

- string
- bytes
- floating
- integer
- mixed-integer
- mixed-integer-float
- decimal
- complex
- categorical
- boolean
- datetime64
- datetime
- date
- timedelta64
- timedelta
- time
- period
- mixed
- unknown-array

**Raises**

- **TypeError** If ndarray-like but cannot infer the dtype
Notes

- 'mixed' is the catchall for anything that is not otherwise specialized
- 'mixed-integer-float' are floats and integers
- 'mixed-integer' are integers mixed with non-integers
- 'unknown-array' is the catchall for something that is an array (has a dtype attribute), but has a dtype unknown to pandas (e.g. external extension array)

Examples

```python
>>> import datetime
>>> infer_dtype(['foo', 'bar'])
'string'

>>> infer_dtype(['a', np.nan, 'b'], skipna=True)
'string'

>>> infer_dtype(['a', np.nan, 'b'], skipna=False)
'mixed'

>>> infer_dtype([b'foo', b'bar'])
'bytes'

>>> infer_dtype([1, 2, 3])
'integer'

>>> infer_dtype([1, 2, 3.5])
'mixed-integer-float'

>>> infer_dtype([1.0, 2.0, 3.5])
'floating'

>>> infer_dtype(['a', 1])
'mixed-integer'

>>> infer_dtype([Decimal('1'), Decimal('2.0')]
'decimal'

>>> infer_dtype([True, False])
'boolean'

>>> infer_dtype([True, False, np.nan])
'boolean'

>>> infer_dtype([pd.Timestamp('20130101')])
'datetime'

>>> infer_dtype([datetime.date(2013, 1, 1)])
'date'
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

```
>>> infer_dtype([np.datetime64('2013-01-01')])
'datetime64'

>>> infer_dtype([datetime.timedelta(0, 1, 1)])
'timedelta'

>>> infer_dtype(pd.Series(list('aabc')).astype('category'))
'categorical'
```

**pandas.api.types.pandas_dtype**

pandas.api.types.pandas_dtype(dtype)

Convert input into a pandas only dtype object or a numpy dtype object.

**Parameters**

dtype [object to be converted]

**Returns**

np.dtype or a pandas dtype

**Raises**

TypeError if not a dtype
### Dtype introspection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api.types.is_bool_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of a boolean dtype.</td>
</tr>
<tr>
<td><code>api.types.is_categorical_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the Categorical dtype.</td>
</tr>
<tr>
<td><code>api.types.is_complex_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of a complex dtype.</td>
</tr>
<tr>
<td><code>api.types.is_datetime64_any_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of the datetime64 dtype.</td>
</tr>
<tr>
<td><code>api.types.is_datetime64_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the datetime64 dtype.</td>
</tr>
<tr>
<td><code>api.types.is_datetime64_ns_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of the datetime64[ns] dtype.</td>
</tr>
<tr>
<td><code>api.types.is_datetime64tz_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of a DatetimeTZDtype dtype.</td>
</tr>
<tr>
<td><code>api.types.is_extension_type(arr)</code></td>
<td>(DEPRECATED) Check whether an array-like is of a pandas extension class instance.</td>
</tr>
<tr>
<td><code>api.types.is_extension_array_dtype(arr_or_dtype)</code></td>
<td>Check if an object is a pandas extension array type.</td>
</tr>
<tr>
<td><code>api.types.is_float_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of a float dtype.</td>
</tr>
<tr>
<td><code>api.types.is_int64_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of the int64 dtype.</td>
</tr>
<tr>
<td><code>api.types.is_integer_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of an integer dtype.</td>
</tr>
<tr>
<td><code>api.types.is_interval_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the Interval dtype.</td>
</tr>
<tr>
<td><code>api.types.is_numeric_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of a numeric dtype.</td>
</tr>
<tr>
<td><code>api.types.is_object_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the object dtype.</td>
</tr>
<tr>
<td><code>api.types.is_period_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the Period dtype.</td>
</tr>
<tr>
<td><code>api.types.is_signed_integer_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of a signed integer dtype.</td>
</tr>
<tr>
<td><code>api.types.is_string_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of the string dtype.</td>
</tr>
<tr>
<td><code>api.types.is_timedelta64_dtype(arr_or_dtype)</code></td>
<td>Check whether an array-like or dtype is of the timedelta64 dtype.</td>
</tr>
<tr>
<td><code>api.types.is_timedelta64_ns_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of the timedelta64[ns] dtype.</td>
</tr>
<tr>
<td><code>api.types.is_unsigned_integer_dtype(arr_or_dtype)</code></td>
<td>Check whether the provided array or dtype is of an unsigned integer dtype.</td>
</tr>
<tr>
<td><code>api.types.is_sparse(arr)</code></td>
<td>Check whether an array-like is a 1-D pandas sparse array.</td>
</tr>
</tbody>
</table>
pandas.api.types.is_bool_dtype

pandas.api.types.is_bool_dtype(arr_or_dtype)
Check whether the provided array or dtype is of a boolean dtype.

Parameters

arr_or_dtype [array-like or dtype] The array or dtype to check.

Returns

boolean Whether or not the array or dtype is of a boolean dtype.

Notes

An ExtensionArray is considered boolean when the _is_boolean attribute is set to True.

Examples

```python
g >>> is_bool_dtype(str)
False
g >>> is_bool_dtype(int)
False
g >>> is_bool_dtype(bool)
True
g >>> is_bool_dtype(np.bool_)
True
g >>> is_bool_dtype(np.array(['a', 'b']))
False
g >>> is_bool_dtype(pd.Series([1, 2]))
False
g >>> is_bool_dtype(np.array([True, False]))
True
g >>> is_bool_dtype(pd.Categorical([True, False]))
True
g >>> is_bool_dtype(pd.arrays.SparseArray([True, False]))
True```

pandas.api.types.is_categorical_dtype

pandas.api.types.is_categorical_dtype(arr_or_dtype)
Check whether an array-like or dtype is of the Categorical dtype.

Parameters

arr_or_dtype [array-like or dtype] The array-like or dtype to check.

Returns

boolean Whether or not the array-like or dtype is of the Categorical dtype.
Examples

```python
>>> is_categorical_dtype(object)
False
>>> is_categorical_dtype(CategoricalDtype())
True
>>> is_categorical_dtype([1, 2, 3])
False
>>> is_categorical_dtype(pd.Categorical([1, 2, 3]))
True
>>> is_categorical_dtype(pd.CategoricalIndex([1, 2, 3]))
True
```

**pandas.api.types.is_complex_dtype**

**pandas.api.types.is_complex_dtype(arr_or_dtype)**
Check whether the provided array or dtype is of a complex dtype.

**Parameters**

- `arr_or_dtype` [array-like or dtype] The array or dtype to check.

**Returns**

- `boolean` Whether or not the array or dtype is of a complex dtype.

Examples

```python
>>> is_complex_dtype(str)
False
>>> is_complex_dtype(int)
False
>>> is_complex_dtype(np.complex_)
True
>>> is_complex_dtype(np.array(['a', 'b']))
False
>>> is_complex_dtype(pd.Series([1, 2]))
False
>>> is_complex_dtype(np.array([1 + 1j, 5]))
True
```

**pandas.api.types.is_datetime64_any_dtype**

**pandas.api.types.is_datetime64_any_dtype(arr_or_dtype)**
Check whether the provided array or dtype is of the datetime64 dtype.

**Parameters**

- `arr_or_dtype` [array-like or dtype] The array or dtype to check.

**Returns**

- `bool` Whether or not the array or dtype is of the datetime64 dtype.
pandas: powerful Python data analysis toolkit, Release 1.4.2

Examples

```python
>>> is_datetime64_any_dtype(str)
False
>>> is_datetime64_any_dtype(int)
False
>>> is_datetime64_any_dtype(np.datetime64)  # can be tz-naive
True
>>> is_datetime64_any_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_any_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_any_dtype(np.array([1, 2]))
False
>>> is_datetime64_any_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_any_dtype(pd.DatetimeIndex([1, 2, 3], dtype="datetime64[ns]"))
True
```

pandas.api.types.is_datetime64_dtype

pandas.api.types.is_datetime64_dtype(arr_or_dtype)

Check whether an array-like or dtype is of the datetime64 dtype.

Parameters

arr_or_dtype  [array-like or dtype] The array-like or dtype to check.

Returns

boolean  Whether or not the array-like or dtype is of the datetime64 dtype.

Examples

```python
>>> is_datetime64_dtype(object)
False
>>> is_datetime64_dtype(np.datetime64)
True
>>> is_datetime64_dtype(np.array([], dtype=int))
False
>>> is_datetime64_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_dtype([1, 2, 3])
False
```
pandas.api.types.is_datetime64_ns_dtype

pandas.api.types.is_datetime64_ns_dtype(arr_or_dtype)
Check whether the provided array or dtype is of the datetime64[ns] dtype.

Parameters

arr_or_dtype [array-like or dtype] The array or dtype to check.

Returns

bool Whether or not the array or dtype is of the datetime64[ns] dtype.

Examples

>>> is_datetime64_ns_dtype(str)
False
>>> is_datetime64_ns_dtype(int)
False
>>> is_datetime64_ns_dtype(np.datetime64) # no unit
False
>>> is_datetime64_ns_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_ns_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_ns_dtype(np.array([1, 2]))
False
>>> is_datetime64_ns_dtype(np.array([], dtype="datetime64[ps]")) # wrong unit
False
>>> is_datetime64_ns_dtype(pd.DatetimeIndex([1, 2, 3], dtype="datetime64[ns]"))
True

pandas.api.types.is_datetime64tz_dtype

pandas.api.types.is_datetime64tz_dtype(arr_or_dtype)
Check whether an array-like or dtype is of a DatetimeTZDtype dtype.

Parameters

arr_or_dtype [array-like or dtype] The array-like or dtype to check.

Returns

boolean Whether or not the array-like or dtype is of a DatetimeTZDtype dtype.
Examples

```python
>>> is_datetime64tz_dtype(object)
False
>>> is_datetime64tz_dtype([1, 2, 3])
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3]))  # tz-naive
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

```python
dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
True
>>> is_datetime64tz_dtype(s)
True
```

### pandas.api.types.is_extension_type

**pandas.api.types.is_extension_type(arr)**

Check whether an array-like is of a pandas extension class instance.

Deprecated since version 1.0.0: Use is_extension_array_dtype instead.

Extension classes include categoricals, pandas sparse objects (i.e. classes represented within the pandas library and not ones external to it like scipy sparse matrices), and datetime-like arrays.

**Parameters**

- **arr** [array-like, scalar] The array-like to check.

**Returns**

- **boolean** Whether or not the array-like is of a pandas extension class instance.

Examples

```python
>>> is_extension_type([1, 2, 3])
False
>>> is_extension_type(np.array([1, 2, 3]))
False
>>> cat = pd.Categorical([1, 2, 3])
>>> is_extension_type(cat)
True
>>> is_extension_type(pd.Series(cat))
True
>>> is_extension_type(pd.arrays.SparseArray([1, 2, 3]))
True
>>> from scipy.sparse import bsr_matrix
>>> is_extension_type(bsr_matrix([1, 2, 3]))
```

(continues on next page)
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_extension_type(s)
True

pandas.api.types.is_extension_array_dtype

pandas.api.types.is_extension_array_dtype(arr_or_dtype)

Check if an object is a pandas extension array type.

See the Use Guide for more.

Parameters

arr_or_dtype [object] For array-like input, the .dtype attribute will be extracted.

Returns

bool Whether the arr_or_dtype is an extension array type.

Notes

This checks whether an object implements the pandas extension array interface. In pandas, this includes:
• Categorical
• Sparse
• Interval
• Period
• DatetimeArray
• TimedeltaArray

Third-party libraries may implement arrays or types satisfying this interface as well.

Examples

>>> from pandas.api.types import is_extension_array_dtype
>>> arr = pd.Categorical(['a', 'b'])
>>> is_extension_array_dtype(arr)
True
>>> is_extension_array_dtype(arr.dtype)
True

>>> arr = np.array(['a', 'b'])
>>> is_extension_array_dtype(arr.dtype)
False

3.14. General utility functions
pandas.api.types.is_float_dtype

pandas.api.types.is_float_dtype(arr_or_dtype)

Check whether the provided array or dtype is of a float dtype.

This function is internal and should not be exposed in the public API.

Parameters

arr_or_dtype [array-like or dtype] The array or dtype to check.

Returns

boolean Whether or not the array or dtype is of a float dtype.

Examples

```python
>>> is_float_dtype(str)
False
>>> is_float_dtype(int)
False
>>> is_float_dtype(float)
True
>>> is_float_dtype(np.array(['a', 'b']))
False
>>> is_float_dtype(pd.Series([1, 2]))
False
>>> is_float_dtype(pd.Index([1, 2.]))
True
```

pandas.api.types.is_int64_dtype

pandas.api.types.is_int64_dtype(arr_or_dtype)

Check whether the provided array or dtype is of the int64 dtype.

Parameters

arr_or_dtype [array-like or dtype] The array or dtype to check.

Returns

boolean Whether or not the array or dtype is of the int64 dtype.

Notes

Depending on system architecture, the return value of is_int64_dtype( int) will be True if the OS uses 64-bit integers and False if the OS uses 32-bit integers.
Examples

```python
>>> is_int64_dtype(str)
False
>>> is_int64_dtype(np.int32)
False
>>> is_int64_dtype(np.int64)
True
>>> is_int64_dtype('int8')
False
>>> is_int64_dtype('Int8')
False
>>> is_int64_dtype(pd.Int64Dtype)
True
>>> is_int64_dtype(float)
False
>>> is_int64_dtype(np.uint64)  # unsigned
False
>>> is_int64_dtype(np.array(['a', 'b']))
False
>>> is_int64_dtype(np.array([1, 2], dtype=np.int64))
True
>>> is_int64_dtype(pd.Index([1, 2.]))  # float
False
>>> is_int64_dtype(np.array([1, 2], dtype=np.uint32))  # unsigned
False
```

pandas.api.types.is_integer_dtype

`pandas.api.types.is_integer_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of an integer dtype.

Unlike in `is_any_int_dtype`, timedelta64 instances will return False.

The nullable Integer dtypes (e.g. pandas.Int64Dtype) are also considered as integer by this function.

**Parameters**

- `arr_or_dtype` [array-like or dtype] The array or dtype to check.

**Returns**

- `boolean` Whether or not the array or dtype is of an integer dtype and not an instance of timedelta64.

**Examples**

```python
>>> is_integer_dtype(str)
False
>>> is_integer_dtype(int)
True
>>> is_integer_dtype(float)
False
>>> is_integer_dtype(np.uint64)
```

(continues on next page)
True

```python
>>> is_integer_dtype('int8')
True
>>> is_integer_dtype('Int8')
True
>>> is_integer_dtype(pd.Int8Dtype)
True
>>> is_integer_dtype(np.datetime64)
False
>>> is_integer_dtype(np.timedelta64)
False
>>> is_integer_dtype(np.array(['a', 'b']))
False
>>> is_integer_dtype(pd.Series([1, 2]))
True
>>> is_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_integer_dtype(pd.Index([1, 2.]))  # float
False
```

**pandas.api.types.is_interval_dtype**

```
pandas.api.types.is_interval_dtype(arr_or_dtype)
    Check whether an array-like or dtype is of the Interval dtype.
    Parameters
        arr_or_dtype [array-like or dtype] The array-like or dtype to check.
    Returns
        boolean  Whether or not the array-like or dtype is of the Interval dtype.
```

**Examples**

```python
>>> is_interval_dtype(object)
False
>>> is_interval_dtype(IntervalDtype())
True
>>> is_interval_dtype([1, 2, 3])
False
>>> interval = pd.Interval(1, 2, closed="right")
>>> is_interval_dtype(interval)
False
>>> is_interval_dtype(pd.IntervalIndex([interval]))
True
```
pandas.api.types.is_numeric_dtype

pandas.api.types.is_numeric_dtype(arr_or_dtype)

Check whether the provided array or dtype is of a numeric dtype.

Parameters

  arr_or_dtype  [array-like or dtype] The array or dtype to check.

Returns

  boolean  Whether or not the array or dtype is of a numeric dtype.

Examples

```python
>>> is_numeric_dtype(str)
False
>>> is_numeric_dtype(int)
True
>>> is_numeric_dtype(float)
True
>>> is_numeric_dtype(np.uint64)
True
>>> is_numeric_dtype(np.datetime64)
False
>>> is_numeric_dtype(np.timedelta64)
False
>>> is_numeric_dtype(np.array(['a', 'b']))
False
>>> is_numeric_dtype(pd.Series([1, 2]))
True
>>> is_numeric_dtype(pd.Index([1, 2.]))
True
>>> is_numeric_dtype(np.array([], dtype=np.timedelta64))
False
```

pandas.api.types.is_object_dtype

pandas.api.types.is_object_dtype(arr_or_dtype)

Check whether an array-like or dtype is of the object dtype.

Parameters

  arr_or_dtype  [array-like or dtype] The array-like or dtype to check.

Returns

  boolean  Whether or not the array-like or dtype is of the object dtype.
Examples

```python
>>> is_object_dtype(object)
True
>>> is_object_dtype(int)
False
>>> is_object_dtype(np.array([], dtype=object))
True
>>> is_object_dtype(np.array([], dtype=int))
False
>>> is_object_dtype([1, 2, 3])
False
```

pandas.api.types.is_period_dtype

```python
pandas.api.types.is_period_dtype(arr_or_dtype)
```

Check whether an array-like or dtype is of the Period dtype.

Parameters

`arr_or_dtype` [array-like or dtype] The array-like or dtype to check.

Returns

boolean Whether or not the array-like or dtype is of the Period dtype.

```python
>>> is_period_dtype(object)
False
>>> is_period_dtype(PeriodDtype(freq="D"))
True
>>> is_period_dtype([1, 2, 3])
False
>>> is_period_dtype(pd.Period("2017-01-01"))
False
>>> is_period_dtype(pd.PeriodIndex([], freq="A"))
True
```

pandas.api.types.is_signed_integer_dtype

```python
pandas.api.types.is_signed_integer_dtype(arr_or_dtype)
```

Check whether the provided array or dtype is of a signed integer dtype. Unlike in `is_any_int_dtype`, timedelta64 instances will return False. The nullable Integer dtypes (e.g. pandas.Int64Dtype) are also considered as integer by this function.

Parameters

`arr_or_dtype` [array-like or dtype] The array or dtype to check.

Returns

boolean Whether or not the array or dtype is of a signed integer dtype and not an instance of timedelta64.

```python
>>> is_signed_integer_dtype(object)
False
>>> is_signed_integer_dtype(PeriodDtype(freq="D"))
True
>>> is_signed_integer_dtype([1, 2, 3])
False
>>> is_signed_integer_dtype(pd.Period("2017-01-01"))
False
>>> is_signed_integer_dtype(pd.PeriodIndex([], freq="A"))
True
```
Examples

```python
>>> is_signed_integer_dtype(str)
False
>>> is_signed_integer_dtype(int)
True
>>> is_signed_integer_dtype(float)
False
>>> is_signed_integer_dtype(np.uint64)  # unsigned
False
>>> is_signed_integer_dtype('int8')
True
>>> is_signed_integer_dtype('Int8')
True
>>> is_signed_integer_dtype(pd.Int8Dtype)
True
>>> is_signed_integer_dtype(np.datetime64)
False
>>> is_signed_integer_dtype(np.timedelta64)
False
>>> is_signed_integer_dtype(np.array(['a', 'b']))
False
>>> is_signed_integer_dtype(pd.Series([1, 2]))
True
>>> is_signed_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_signed_integer_dtype(pd.Index([1, 2.]))  # float
False
>>> is_signed_integer_dtype(np.array([1, 2], dtype=np.uint32))  # unsigned
False
```

**pandas.api.types.is_string_dtype**

pandas.api.types.is_string_dtype(arr_or_dtype)

Check whether the provided array or dtype is of the string dtype.

**Parameters**

- **arr_or_dtype** [array-like or dtype] The array or dtype to check.

**Returns**

- **boolean** Whether or not the array or dtype is of the string dtype.
Examples

```python
>>> is_string_dtype(str)
True
>>> is_string_dtype(object)
True
>>> is_string_dtype(int)
False
>>> is_string_dtype(np.array(['a', 'b']))
True
>>> is_string_dtype(pd.Series([1, 2]))
False
```

**pandas.api.types.is_timedelta64_dtype**

pandas.api.types.is_timedelta64_dtype(arr_or_dtype)

Check whether an array-like or dtype is of the timedelta64 dtype.

**Parameters**

- arr_or_dtype [array-like or dtype] The array-like or dtype to check.

**Returns**

- boolean Whether or not the array-like or dtype is of the timedelta64 dtype.

Examples

```python
>>> is_timedelta64_dtype(object)
False
>>> is_timedelta64_dtype(np.timedelta64)
True
>>> is_timedelta64_dtype([1, 2, 3])
False
>>> is_timedelta64_dtype(pd.Series([], dtype="timedelta64[ns]"))
True
>>> is_timedelta64_dtype('0 days')
False
```

**pandas.api.types.is_timedelta64_ns_dtype**

pandas.api.types.is_timedelta64_ns_dtype(arr_or_dtype)

Check whether the provided array or dtype is of the timedelta64[ns] dtype.

This is a very specific dtype, so generic ones like np.timedelta64 will return False if passed into this function.

**Parameters**

- arr_or_dtype [array-like or dtype] The array or dtype to check.

**Returns**

- boolean Whether or not the array or dtype is of the timedelta64[ns] dtype.
Examples

```python
>>> is_timedelta64_ns_dtype(np.dtype('m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.dtype('m8[ps]'))  # Wrong frequency
False
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype='m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype=np.timedelta64))
False
```

pandas.api.types.is_unsigned_integer_dtype

```
pandas.api.types.is_unsigned_integer_dtype(arr_or_dtype)

Check whether the provided array or dtype is of an unsigned integer dtype.

Parameters

dtype [array-like or dtype] The array or dtype to check.

Returns

boolean Whether or not the array or dtype is of an unsigned integer dtype.
```

Examples

```python
>>> is_unsigned_integer_dtype(str)
False
>>> is_unsigned_integer_dtype(int)  # signed
False
>>> is_unsigned_integer_dtype(float)
False
>>> is_unsigned_integer_dtype(np.uint64)
True
>>> is_unsigned_integer_dtype('uint8')
True
>>> is_unsigned_integer_dtype('UInt8')
True
>>> is_unsigned_integer_dtype(pd.UInt8Dtype)
True
>>> is_unsigned_integer_dtype(np.array(['a', 'b']))
False
>>> is_unsigned_integer_dtype(pd.Series([1, 2]))  # signed
False
>>> is_unsigned_integer_dtype(pd.Index([1, 2.]))  # float
False
>>> is_unsigned_integer_dtype(np.array([1, 2], dtype=np.uint32))
True
```
pandas.api.types.is_sparse

pandas.api.types.is_sparse(arr)
Check whether an array-like is a 1-D pandas sparse array.
Check that the one-dimensional array-like is a pandas sparse array. Returns True if it is a pandas sparse array, not another type of sparse array.

Parameters

arr [array-like] Array-like to check.

Returns

bool Whether or not the array-like is a pandas sparse array.

Examples

Returns True if the parameter is a 1-D pandas sparse array.

```python
>>> is_sparse(pd.arrays.SparseArray([0, 0, 1, 0]))
True
>>> is_sparse(pd.Series(pd.arrays.SparseArray([0, 0, 1, 0])))
True
```

Returns False if the parameter is not sparse.

```python
>>> is_sparse(np.array([0, 0, 1, 0]))
False
>>> is_sparse(pd.Series([0, 1, 0, 0]))
False
```

Returns False if the parameter is not a pandas sparse array.

```python
>>> from scipy.sparse import bsr_matrix
>>> is_sparse(bsr_matrix([0, 1, 0, 0]))
False
```

Returns False if the parameter has more than one dimension.

Iterable introspection

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>api.types.is_dict_like(obj)</td>
<td>Check if the object is dict-like.</td>
</tr>
<tr>
<td>api.types.is_file_like(obj)</td>
<td>Check if the object is a file-like object.</td>
</tr>
<tr>
<td>api.types.is_list_like</td>
<td>Check if the object is list-like.</td>
</tr>
<tr>
<td>api.types.is_named_tuple(obj)</td>
<td>Check if the object is a named tuple.</td>
</tr>
<tr>
<td>api.types.is_iterator</td>
<td>Check if the object is an iterator.</td>
</tr>
</tbody>
</table>
pandas.api.types.is_dict_like

pandas.api.types.is_dict_like(obj)

Check if the object is dict-like.

Parameters

obj [The object to check]

Returns

is_dict_like [bool] Whether obj has dict-like properties.

Examples

```python
>>> is_dict_like({1: 2})
True
>>> is_dict_like([1, 2, 3])
False
>>> is_dict_like(dict)
False
>>> is_dict_like(dict())
True
```

pandas.api.types.is_file_like

pandas.api.types.is_file_like(obj)

Check if the object is a file-like object.

For objects to be considered file-like, they must be an iterator AND have either a read and/or write method as an attribute.

Note: file-like objects must be iterable, but iterable objects need not be file-like.

Parameters

obj [The object to check]

Returns

is_file_like [bool] Whether obj has file-like properties.

Examples

```python
>>> import io
>>> buffer = io.StringIO("data")
>>> is_file_like(buffer)
True
>>> is_file_like([1, 2, 3])
False
```
pandas.api.types.is_list_like

pandas.api.types.is_list_like()  
Check if the object is list-like.  
Objects that are considered list-like are for example Python lists, tuples, sets, NumPy arrays, and Pandas Series.  
Strings and datetime objects, however, are not considered list-like.  

Parameters

- obj [object] Object to check.
- allow_sets [bool, default True] If this parameter is False, sets will not be considered list-like.

Returns

- bool Whether obj has list-like properties.

Examples

```python
>>> import datetime
>>> is_list_like([1, 2, 3])
True
>>> is_list_like({1, 2, 3})
True
>>> is_list_like(datetime.datetime(2017, 1, 1))
False
>>> is_list_like("foo")
False
>>> is_list_like(1)
False
>>> is_list_like(np.array([2]))
True
>>> is_list_like(np.array(2))
False
```

pandas.api.types.is_named_tuple

pandas.api.types.is_named_tuple(obj)  
Check if the object is a named tuple.

Parameters

- obj [The object to check]

Returns

- is_named_tuple [bool] Whether obj is a named tuple.
Examples

```python
>>> from collections import namedtuple
>>> Point = namedtuple("Point", ["x", "y"])
>>> p = Point(1, 2)
>>> is_named_tuple(p)
True
>>> is_named_tuple((1, 2))
False
```

docstring:

pandas.api.types.is_iterator

docstring:

pandas.api.types.is_iterator()

Check if the object is an iterator.

This is intended for generators, not list-like objects.

Parameters

- **obj** [The object to check]

Returns

- **is_iter** [bool] Whether *obj* is an iterator.

Examples

```python
>>> import datetime

>>> is_iterator((x for x in []))
True
>>> is_iterator([1, 2, 3])
False
>>> is_iterator(datetime.datetime(2017, 1, 1))
False
>>> is_iterator("foo")
False
>>> is_iterator(1)
False
```
 Scalar introspection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api.types.is_bool</code></td>
<td>Return True if given object is boolean.</td>
</tr>
<tr>
<td><code>api.types.is_categorical(arr)</code></td>
<td>Check whether an array-like is a Categorical instance.</td>
</tr>
<tr>
<td><code>api.types.is_complex</code></td>
<td>Return True if given object is complex.</td>
</tr>
<tr>
<td><code>api.types.is_float</code></td>
<td>Return True if given object is float.</td>
</tr>
<tr>
<td><code>api.types.is_hashable(obj)</code></td>
<td>Return True if hash(obj) will succeed, False otherwise.</td>
</tr>
<tr>
<td><code>api.types.is_integer</code></td>
<td>Return True if given object is integer.</td>
</tr>
<tr>
<td><code>api.types.is_interval</code></td>
<td></td>
</tr>
<tr>
<td><code>api.types.is_number(obj)</code></td>
<td>Check if the object is a number.</td>
</tr>
<tr>
<td><code>api.types.is_re(obj)</code></td>
<td>Check if the object is a regex pattern instance.</td>
</tr>
<tr>
<td><code>api.types.is_re_compilable(obj)</code></td>
<td>Check if the object can be compiled into a regex pattern instance.</td>
</tr>
<tr>
<td><code>api.types.is_scalar</code></td>
<td>Return True if given object is scalar.</td>
</tr>
</tbody>
</table>

**pandas.api.types.is_bool**

```python
pandas.api.types.is_bool()  
Return True if given object is boolean.

Returns

bool
```

**pandas.api.types.is_categorical**

```python
pandas.api.types.is_categorical(arr)  
Check whether an array-like is a Categorical instance.

Parameters

- `arr` [array-like] The array-like to check.

Returns

boolean Whether or not the array-like is of a Categorical instance.
```

**Examples**

```python
>>> is_categorical([1, 2, 3])
False
```

Categoricals, Series Categoricals, and CategoricalIndex will return True.

```python
>>> cat = pd.Categorical([1, 2, 3])
>>> is_categorical(cat)
True
>>> is_categorical(pd.Series(cat))
True
>>> is_categorical(pd.CategoricalIndex([1, 2, 3]))
True
```
pandas.api.types.is_complex

pandas.api.types.is_complex()

Return True if given object is complex.

Returns

bool

pandas.api.types.is_float

pandas.api.types.is_float()

Return True if given object is float.

Returns

bool

pandas.api.types.is_hashable

pandas.api.types.is_hashable(obj)

Return True if hash(obj) will succeed, False otherwise.

Some types will pass a test against collections.abc.Hashable but fail when they are actually hashed with hash().

Distinguish between these and other types by trying the call to hash() and seeing if they raise TypeError.

Returns

bool

Examples

```python
>>> import collections
>>> a = ([],)
>>> isinstance(a, collections.abc.Hashable)
True
>>> is_hashable(a)
False
```

pandas.api.types.is_integer

pandas.api.types.is_integer()

Return True if given object is integer.

Returns

bool
**pandas.api.types.is_interval**

`pandas.api.types.is_interval()`

**pandas.api.types.is_number**

`pandas.api.types.is_number(obj)`

Check if the object is a number.

Returns True when the object is a number, and False if not.

**Parameters**

- **obj** [any type] The object to check if is a number.

**Returns**

- **is_number** [bool] Whether `obj` is a number or not.

**See also:**

`api.types.is_integer` Checks a subgroup of numbers.

**Examples**

```python
>>> from pandas.api.types import is_number
>>> is_number(1)
True
>>> is_number(7.15)
True
```

Booleans are valid because they are int subclass.

```python
>>> is_number(False)
True
>>> is_number("foo")
False
>>> is_number("5")
False
```

**pandas.api.types.is_re**

`pandas.api.types.is_re(obj)`

Check if the object is a regex pattern instance.

**Parameters**

- **obj** [The object to check]

**Returns**

- **is_re** [bool] Whether `obj` is a regex pattern.
Examples

```python
>>> is_re(re.compile(".*"))
True
>>> is_re("foo")
False
```

**pandas.api.types.is_re_compilable**

pandas.api.types.is_re_compilable(obj)

Check if the object can be compiled into a regex pattern instance.

**Parameters**

- obj [The object to check]

**Returns**

- is_regex_compilable [bool] Whether obj can be compiled as a regex pattern.

Examples

```python
>>> is_re_compilable(".*")
True
>>> is_re_compilable(1)
False
```

**pandas.api.types.is_scalar**

pandas.api.types.is_scalar()

Return True if given object is scalar.

**Parameters**

- val [object] This includes:
  - numpy array scalar (e.g. np.int64)
  - Python builtin numerics
  - Python builtin byte arrays and strings
  - None
  - datetime.datetime
  - datetime.timedelta
  - Period
  - decimal.Decimal
  - Interval
  - DateOffset
  - Fraction
  - Number.
Returns

bool  Return True if given object is scalar.

Examples

```python
>>> import datetime
>>> dt = datetime.datetime(2018, 10, 3)
>>> pd.api.types.is_scalar(dt)
True

>>> pd.api.types.is_scalar([2, 3])
False

>>> pd.api.types.is_scalar({0: 1, 2: 3})
False

>>> pd.api.types.is_scalar((0, 2))
False
```

pandas supports PEP 3141 numbers:

```python
>>> from fractions import Fraction
>>> pd.api.types.is_scalar(Fraction(3, 5))
True
```

3.14.5 Bug report function

`show_versions([as_json])` Provide useful information, important for bug reports.

**pandas.show_versions**

`pandas.show_versions(as_json=False)` Provide useful information, important for bug reports.

It comprises info about hosting operation system, pandas version, and versions of other installed relative packages.

**Parameters**

`as_json` [str or bool, default False]

- If False, outputs info in a human readable form to the console.
- If str, it will be considered as a path to a file. Info will be written to that file in JSON format.
- If True, outputs info in JSON format to the console.
3.15 Extensions

These are primarily intended for library authors looking to extend pandas objects.

<table>
<thead>
<tr>
<th>api.extensions.register_extension_dtype(cls)</th>
<th>Register an ExtensionType with pandas as class decorator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>api.extensions.register_dataframe_accessor(name)</td>
<td>Register a custom accessor on DataFrame objects.</td>
</tr>
<tr>
<td>api.extensions.register_series_accessor(name)</td>
<td>Register a custom accessor on Series objects.</td>
</tr>
<tr>
<td>api.extensions.register_index_accessor(name)</td>
<td>Register a custom accessor on Index objects.</td>
</tr>
<tr>
<td>api.extensions.ExtensionDtype()</td>
<td>A custom data type, to be paired with an ExtensionArray.</td>
</tr>
</tbody>
</table>

3.15.1 pandas.api.extensions.register_extension_dtype

pandas.api.extensions.register_extension_dtype(cls)

Register an ExtensionType with pandas as class decorator.

This enables operations like .astype(name) for the name of the ExtensionDtype.

Returns:

callable A class decorator.

Examples

```python
>>> from pandas.api.extensions import register_extension_dtype, ExtensionDtype
>>> @register_extension_dtype
... class MyExtensionDtype(ExtensionDtype):
...     name = "myextension"
```

3.15.2 pandas.api.extensions.register_dataframe_accessor

pandas.api.extensions.register_dataframe_accessor(name)

Register a custom accessor on DataFrame objects.

Parameters:

name [str] Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

Returns:

callable A class decorator.

See also:

register_dataframe_accessor Register a custom accessor on DataFrame objects.
register_series_accessor Register a custom accessor on Series objects.
register_index_accessor Register a custom accessor on Index objects.
Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```python
def __init__(self, pandas_object):  # noqa: E999
...
```

For consistency with pandas methods, you should raise an AttributeError if the data passed to your accessor has an incorrect dtype.

```python
>>> pd.Series(["a", "b"]).dt
Traceback (most recent call last):
... AttributeError: Can only use .dt accessor with datetimelike values
```

Examples

In your library code:

```python
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Back in an interactive IPython session:

```python
In [1]: ds = pd.DataFrame({"longitude": np.linspace(0, 10), ...
                     "latitude": np.linspace(0, 20)})
In [2]: ds.geo.center
Out[2]: (5.0, 10.0)
In [3]: ds.geo.plot()  # plots data on a map
```
3.15.3 pandas.api.extensions.register_series_accessor

```python
def __init__(self, pandas_object):  # noqa: E999
    ... 
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```python
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

Examples

In your library code:

```python
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
```
Back in an interactive IPython session:

```
In [1]: ds = pd.DataFrame({"longitude": np.linspace(0, 10),
...:                     "latitude": np.linspace(0, 20)})
In [2]: ds.geo.center
Out[2]: (5.0, 10.0)
In [3]: ds.geo.plot()  # plots data on a map
```

### 3.15.4 pandas.api.extensions.register_index_accessor

`pandas.api.extensions.register_index_accessor(name)`

Register a custom accessor on Index objects.

**Parameters**

- `name` [str] Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

**Returns**

`callable` A class decorator.

**See also:**

- `register_dataframe_accessor` Register a custom accessor on DataFrame objects.
- `register_series_accessor` Register a custom accessor on Series objects.
- `register_index_accessor` Register a custom accessor on Index objects.

**Notes**

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):  # noqa: E999
...  
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(["a", "b"]).dt
Traceback (most recent call last):
  ...
AttributeError: Can only use .dt accessor with datetimelike values
```
Examples

In your library code:

```python
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass

Back in an interactive IPython session:

```
In [1]: ds = pd.DataFrame({"longitude": np.linspace(0, 10),
                        "latitude": np.linspace(0, 20)})
In [2]: ds.geo.center
Out[2]: (5.0, 10.0)
In [3]: ds.geo.plot()  # plots data on a map
```

3.15.5 pandas.api.extensions.ExtensionDtype

class pandas.api.extensions.ExtensionDtype

A custom data type, to be paired with an ExtensionArray.

See also:

- `extensions.register_extension_dtype` Register an ExtensionType with pandas as class decorator.
- `extensions.ExtensionArray` Abstract base class for custom 1-D array types.

Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- `type`
- `name`
- `construct_array_type`

The following attributes and methods influence the behavior of the dtype in pandas operations:

- `is_numeric`
- `is_boolean`
- `get_common_dtype`

The `na_value` class attribute can be used to set the default NA value for this type. `numpy.nan` is used by default.
ExtensionDtypes are required to be hashable. The base class provides a default implementation, which relies on the _metadata class attribute. _metadata should be a tuple containing the strings that define your data type. For example, with PeriodDtype that’s the freq attribute.

If you have a parametrized dtype you should set the `_metadata` class property.

Ideally, the attributes in _metadata will match the parameters to your ExtensionDtype.__init__ (if any). If any of the attributes in _metadata don’t implement the standard __eq__ or __hash__, the default implementations here will not work.

For interaction with Apache Arrow (pyarrow), a __from_arrow__ method can be implemented: this method receives a pyarrow Array or ChunkedArray as only argument and is expected to return the appropriate pandas ExtensionArray for this dtype and the passed values:

```python
class ExtensionDtype:
    def __from_arrow__(
        self, array: Union[pyarrow.Array, pyarrow.ChunkedArray]
    ) -> ExtensionArray:
        ...
```

This class does not inherit from ‘abc.ABCMeta’ for performance reasons. Methods and properties required by the interface raise pandas.errors(AbstractMethodError and no register method is provided for registering virtual subclasses.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>A character code (one of ‘biufcmMOSUV’), default ‘O’</td>
</tr>
<tr>
<td>na_value</td>
<td>Default NA value to use for this type.</td>
</tr>
<tr>
<td>name</td>
<td>A string identifying the data type.</td>
</tr>
<tr>
<td>names</td>
<td>Ordered list of field names, or None if there are no fields.</td>
</tr>
<tr>
<td>type</td>
<td>The scalar type for the array, e.g.</td>
</tr>
</tbody>
</table>

pandas.api.extensions.ExtensionDtype.kind

property ExtensionDtype.kind

A character code (one of ‘biufcmMOSUV’), default ‘O’

This should match the NumPy dtype used when the array is converted to an ndarray, which is probably ‘O’ for object if the extension type cannot be represented as a built-in NumPy type.

See also:

numpy.dtype.kind
pandas.api.extensions.ExtensionDtype.na_value

```python
property pandas.api.extensions.ExtensionDtype.na_value

Default NA value to use for this type.

This is used in e.g. ExtensionArray.take. This should be the user-facing “boxed” version of the NA value, not the physical NA value for storage. e.g. for JSONArray, this is an empty dictionary.
```

pandas.api.extensions.ExtensionDtype.name

```python
property pandas.api.extensions.ExtensionDtype.name

A string identifying the data type.

Will be used for display in, e.g. Series.dtype
```

pandas.api.extensions.ExtensionDtype.names

```python
property pandas.api.extensions.ExtensionDtype.names

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.
```

pandas.api.extensions.ExtensionDtype.type

```python
property pandas.api.extensions.ExtensionDtype.type

The scalar type for the array, e.g. int

It's expected ExtensionArray[item] returns an instance of ExtensionDtype.type for scalar item, assuming that value is valid (not NA). NA values do not need to be instances of type.
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>construct_array_type()</td>
<td>Return the array type associated with this dtype.</td>
</tr>
<tr>
<td>construct_from_string(string)</td>
<td>Construct this type from a string.</td>
</tr>
<tr>
<td>empty(shape)</td>
<td>Construct an ExtensionArray of this dtype with the given shape.</td>
</tr>
<tr>
<td>is_dtype(dtype)</td>
<td>Check if we match 'dtype'.</td>
</tr>
</tbody>
</table>

pandas.api.extensions.ExtensionDtype.construct_array_type

```python
classmethod pandas.api.extensions.ExtensionDtype.construct_array_type()

Return the array type associated with this dtype.

Returns

type
```
pandas.api.extensions.ExtensionDtype.construct_from_string

classmethod ExtensionDtype.construct_from_string(string)

Construct this type from a string.

This is useful mainly for data types that accept parameters. For example, a period dtype accepts a frequency parameter that can be set as period[H] (where H means hourly frequency).

By default, in the abstract class, just the name of the type is expected. But subclasses can overwrite this method to accept parameters.

Parameters

string [str] The name of the type, for example category.

Returns

ExtensionDtype Instance of the dtype.

Raises

TypeError If a class cannot be constructed from this 'string'.

Examples

For extension dtypes with arguments the following may be an adequate implementation.

```python
>>> @classmethod
... def construct_from_string(cls, string):
...     pattern = re.compile(r"^my_type\[(?P<arg_name>.+)\]$")
...     match = pattern.match(string)
...     if match:
...         return cls(**match.groupdict())
...     else:
...         raise TypeError("Cannot construct a '{cls.__name__}' from '{string}'")
```

pandas.api.extensions.ExtensionDtype.empty

ExtensionDtype.empty(shape)

Construct an ExtensionArray of this dtype with the given shape.

Analogous to numpy.empty.

Parameters

shape [int or tuple[int]]

Returns

ExtensionArray
pandas.api.extensions.ExtensionDtype.is_dtype

**classmethod** ExtensionDtype.is_dtype(dytype)

Check if we match ‘dtype’.

**Parameters**

- **dtype** [object] The object to check.

**Returns**

- **bool**

**Notes**

The default implementation is True if

1. cls.construct_from_string(dtype) is an instance of cls.
2. dtype is an object and is an instance of cls
3. dtype has a dtype attribute, and any of the above conditions is true for dtype.dtype.

---

api.extensions.ExtensionArray()

Abstract base class for custom 1-D array types.

arrays.PandasArray(values[, copy])

A pandas ExtensionArray for NumPy data.

---

3.15.6 pandas.api.extensions.ExtensionArray

**class** pandas.api.extensions_EXTENSIONArray

Abstract base class for custom 1-D array types.

pandas will recognize instances of this class as proper arrays with a custom type and will not attempt to coerce them to objects. They may be stored directly inside a DataFrame or Series.

**Notes**

The interface includes the following abstract methods that must be implemented by subclasses:

- _from_sequence
- _from_factorized
- __getitem__
- __len__
- __eq__
- dtype
- nbytes
- isna
- take
- copy
- _concat_same_type

A default repr displaying the type, (truncated) data, length, and dtype is provided. It can be customized or replaced by by overriding:

- __repr__: A default repr for the ExtensionArray.
- _formatter: Print scalars inside a Series or DataFrame.

Some methods require casting the ExtensionArray to an ndarray of Python objects with self.astype(object), which may be expensive. When performance is a concern, we highly recommend overriding the following methods:
• fillna
• dropna
• unique
• factorize / _values_for_factorize
• argsort / _values_for_argsort
• searchsorted

The remaining methods implemented on this class should be performant, as they only compose abstract methods. Still, a more efficient implementation may be available, and these methods can be overridden.

One can implement methods to handle array reductions.
• _reduce

One can implement methods to handle parsing from strings that will be used in methods such as pandas.io.parsers.read_csv.
• _from_sequence_of_strings

This class does not inherit from ‘abc.ABCMeta’ for performance reasons. Methods and properties required by the interface raise pandas.errors.AbstractMethodError and no register method is provided for registering virtual subclasses.

ExtensionArrays are limited to 1 dimension.

They may be backed by none, one, or many NumPy arrays. For example, pandas.Categorical is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 address may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists. Pandas makes no assumptions on how the data are stored, just that it can be converted to a NumPy array. The ExtensionArray interface does not impose any rules on how this data is stored. However, currently, the backing data cannot be stored in attributes called .values or ._values to ensure full compatibility with pandas internals. But other names as .data, ._data, ._items, ... can be freely used.

If implementing NumPy’s __array_ufunc__ interface, pandas expects that
1. You defer by returning NotImplemented when any Series are present in inputs. Pandas will extract the arrays and call the ufunc again.
2. You define a _HANDLED_TYPES tuple as an attribute on the class. Pandas inspect this to determine whether the ufunc is valid for the types present.

See NumPy universal functions for more.

By default, ExtensionArrays are not hashable. Immutable subclasses may override this behavior.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype</td>
<td>An instance of 'ExtensionDtype'.</td>
</tr>
<tr>
<td>nbytes</td>
<td>The number of bytes needed to store this object in memory.</td>
</tr>
<tr>
<td>ndim</td>
<td>Extension Arrays are only allowed to be 1-dimensional.</td>
</tr>
<tr>
<td>shape</td>
<td>Return a tuple of the array dimensions.</td>
</tr>
</tbody>
</table>
pandas.api.extensions.ExtensionArray.dtype

**property** `ExtensionArray.dtype`

An instance of ‘ExtensionDtype’.

pandas.api.extensions.ExtensionArray.nbytes

**property** `ExtensionArray.nbytes`

The number of bytes needed to store this object in memory.

pandas.api.extensions.ExtensionArray.ndim

**property** `ExtensionArray.ndim`

Extension Arrays are only allowed to be 1-dimensional.

pandas.api.extensions.ExtensionArray.shape

**property** `ExtensionArray.shape`

Return a tuple of the array dimensions.
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argsort([ascending, kind, na_position])</code></td>
<td>Return the indices that would sort this array.</td>
</tr>
<tr>
<td><code>astype(dtype[, copy])</code></td>
<td>Cast to a NumPy array or ExtensionArray with <code>dtype</code>.</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td>Return a copy of the array.</td>
</tr>
<tr>
<td><code>dropna()</code></td>
<td>Return ExtensionArray without NA values.</td>
</tr>
<tr>
<td><code>factorize([na_sentinel])</code></td>
<td>Encode the extension array as an enumerated type.</td>
</tr>
<tr>
<td><code>fillna([value, method, limit])</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>equals(other)</code></td>
<td>Return if another array is equivalent to this array.</td>
</tr>
<tr>
<td><code>insert(loc, item)</code></td>
<td>Insert an item at the given position.</td>
</tr>
<tr>
<td><code>isin(values)</code></td>
<td>Pointwise comparison for set containment in the given values.</td>
</tr>
<tr>
<td><code>isna()</code></td>
<td>A 1-D array indicating if each value is missing.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return a flattened view on this array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of a ExtensionArray.</td>
</tr>
<tr>
<td><code>searchsorted(value[, side, sorter])</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>shift([periods, fill_value])</code></td>
<td>Shift values by desired number.</td>
</tr>
<tr>
<td><code>take(indices, *[allow_fill, fill_value])</code></td>
<td>Take elements from an array.</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return a list of the values.</td>
</tr>
<tr>
<td><code>unique()</code></td>
<td>Compute the ExtensionArray of unique values.</td>
</tr>
<tr>
<td><code>view([dtype])</code></td>
<td>Return a view on the array.</td>
</tr>
<tr>
<td><code>_concat_same_type(to_concat)</code></td>
<td>Concatenate multiple array of this dtype.</td>
</tr>
<tr>
<td><code>_formatter([boxed])</code></td>
<td>Formatting function for scalar values.</td>
</tr>
<tr>
<td><code>_from_factorized(values, original)</code></td>
<td>Reconstruct an ExtensionArray after factorization.</td>
</tr>
<tr>
<td><code>_from_sequence(scalars, *[dtype, copy])</code></td>
<td>Construct a new ExtensionArray from a sequence of scalars.</td>
</tr>
<tr>
<td><code>_from_sequence_of_strings(strings, *[...])</code></td>
<td>Construct a new ExtensionArray from a sequence of strings.</td>
</tr>
<tr>
<td><code>_reduce(name, *[skipna])</code></td>
<td>Return a scalar result of performing the reduction operation.</td>
</tr>
<tr>
<td><code>_values_for_argsort()</code></td>
<td>Return values for sorting.</td>
</tr>
<tr>
<td><code>_values_for_factorize()</code></td>
<td>Return an array and missing value suitable for factorization.</td>
</tr>
</tbody>
</table>

#### pandas.api.extensions.ExtensionArray.argsort

ExtensionArray.argsort(ascending=True, kind='quicksort', na_position='last', *args, **kwargs)

Return the indices that would sort this array.

**Parameters**

- `ascending` [bool, default True] Whether the indices should result in an ascending or descending sort.
- `*args, **kwargs`: Passed through to `numpy.argsort()`.

**Returns**

np.ndarray[np.intp] Array of indices that sort `self`. If NaN values are contained, NaN values are placed at the end.
See also:

- `numpy.argsort` Sorting implementation used internally.

**pandas.api.extensions.ExtensionArray.astype**

`ExtensionArray.astype(dtype, copy=True)`

Cast to a NumPy array or ExtensionArray with ‘dtype’.

**Parameters**

- **dtype** [str or dtype] Typecode or data-type to which the array is cast.
- **copy** [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

**Returns**

- **array** [np.ndarray or ExtensionArray] An ExtensionArray if dtype is ExtensionDtype, Otherwise a NumPy ndarray with ‘dtype’ for its dtype.

**pandas.api.extensions.ExtensionArray.copy**

`ExtensionArray.copy()`

Return a copy of the array.

**Returns**

- **ExtensionArray**

**pandas.api.extensions.ExtensionArray.dropna**

`ExtensionArray.dropna()`

Return ExtensionArray without NA values.

**Returns**

- **valid** [ExtensionArray]

**pandas.api.extensions.ExtensionArray.factorize**

`ExtensionArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

**Parameters**

- **na_sentinel** [int, default -1] Value to use in the codes array to indicate missing values.

**Returns**

- **codes** [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.
uniques [ExtensionArray] An ExtensionArray containing the unique values of self.

Note: uniques will not contain an entry for the NA value of the ExtensionArray if there are any missing values present in self.

See also:

factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a sort keyword as well.

pandas.api.extensions.ExtensionArray.fillna

ExtensionArray.fillna(value=None, method=None, limit=None)

Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap.

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

ExtensionArray With NA/NaN filled.

pandas.api.extensions.ExtensionArray.equals

ExtensionArray.equals(other)

Return if another array is equivalent to this array.

Equivalent means that both arrays have the same shape and dtype, and all values compare equal. Missing values in the same location are considered equal (in contrast with normal equality).

Parameters

other [ExtensionArray] Array to compare to this Array.

Returns

boolean Whether the arrays are equivalent.
pandas.api.extensions.ExtensionArray.insert

ExtensionArray.insert(loc, item)
Insert an item at the given position.

Parameters
- loc [int]
- item [scalar-like]

Returns
same type as self

Notes
This method should be both type and dtype-preserving. If the item cannot be held in an array of this type/dtype, either ValueError or TypeError should be raised.
The default implementation relies on _from_sequence to raise on invalid items.

pandas.api.extensions.ExtensionArray.isin

ExtensionArray.isin(values)
Pointwise comparison for set containment in the given values.
Roughly equivalent to np.array([x in values for x in self])

Parameters
- values [Sequence]

Returns
np.ndarray[bool]

pandas.api.extensions.ExtensionArray.isna

ExtensionArray.isna()
A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like SparseArray, where returning an ndarray would be expensive, an ExtensionArray may be returned.
Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- `na_values` should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.api.extensions.ExtensionArray.ravel

`ExtensionArray.ravel(order='C')`

Return a flattened view on this array.

Parameters


Returns

- `ExtensionArray`

Notes

- Because ExtensionArrays are 1D-only, this is a no-op.
- The “order” argument is ignored, is for compatibility with NumPy.

pandas.api.extensions.ExtensionArray.repeat

`ExtensionArray.repeat(repeats, axis=None)`

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

- `repeats` [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.
- `axis` [None] Must be None. Has no effect but is accepted for compatibility with numpy.

Returns

- `repeated_array` [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

- `Series.repeat` Equivalent function for Series.
- `Index.repeat` Equivalent function for Index.
- `numpy.repeat` Similar method for `numpy.ndarray`.
- `ExtensionArray.take` Take arbitrary positions.
Examples

```python
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
['a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> cat.repeat(2)
['a', 'a', 'b', 'b', 'c', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> cat.repeat([1, 2, 3])
['a', 'b', 'b', 'c', 'c', 'c']
Categories (3, object): ['a', 'b', 'c']
```

## pandas.api.extensions.ExtensionArray.searchsorted

ExtensionArray.searchsorted(value, side='left', sorter=None)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array `self` (a) such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

Assuming that `self` is sorted:

<table>
<thead>
<tr>
<th>side</th>
<th>returned index / satisfies</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td><code>self[i-1] &lt; value &lt;= self[i]</code></td>
</tr>
<tr>
<td>right</td>
<td><code>self[i-1] &lt;= value &lt; self[i]</code></td>
</tr>
</tbody>
</table>

**Parameters**

- `value` [array-like, list or scalar] Value(s) to insert into `self`.
- `side` [{‘left’, ‘right’}, optional] If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).
- `sorter` [1-D array-like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

**Returns**

- array of ints or int If value is array-like, array of insertion points. If value is scalar, a single integer.

**See also:**

- `numpy.searchsorted` Similar method from NumPy.
pandas.api.extensions.ExtensionArray.shift

ExtensionArray.shift(periods=1, fill_value=None)
Shift values by desired number.
Newly introduced missing values are filled with self.dtype.na_value.

Parameters

- **periods** [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.
- **fill_value** [object, optional] The scalar value to use for newly introduced missing values. The default is self.dtype.na_value.

Returns

- ExtensionArray Shifted.

Notes

- If self is empty or periods is 0, a copy of self is returned.
- If periods > len(self), then an array of size len(self) is returned, with all values filled with self.dtype.na_value.

pandas.api.extensions.ExtensionArray.take

ExtensionArray.take(indices, *, allow_fill=False, fill_value=None)
Take elements from an array.

Parameters

- **indices** [sequence of int or one-dimensional np.ndarray of int] Indices to be taken.
- **allow_fill** [bool, default False] How to handle negative values in indices.
  - False: negative values in indices indicate positional indices from the right (the default). This is similar to numpy.take().
  - True: negative values in indices indicate missing values. These values are set to fill_value. Any other other negative values raise a ValueError.
- **fill_value** [any, optional] Fill value to use for NA-indices when allow_fill is True. This may be None, in which case the default NA value for the type, self.dtype.na_value, is used.

For many ExtensionArrays, there will be two representations of fill_value: a user-facing “boxed” scalar, and a low-level physical NA value. fill_value should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

- ExtensionArray

Raises

- IndexError When the indices are out of bounds for the array.
- ValueError When indices contains negative values other than -1 and allow_fill is True.
See also:

- numpy.take: Take elements from an array along an axis.
- api.extensions.take: Take elements from an array.

Notes

`ExtensionArray.take` is called by `Series.__getitem__`, `.loc`, `.iloc`, when `indices` is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a `fill_value`.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```python
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

**pandas.api.extensions.ExtensionArray.tolist**

`ExtensionArray.tolist()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

Returns

list
pandas.api.extensions.ExtensionArray.unique

ExtensionArray.unique()
Compute the ExtensionArray of unique values.

Returns

uniques  [ExtensionArray]

pandas.api.extensions.ExtensionArray.view

ExtensionArray.view(dtype=None)
Return a view on the array.

Parameters

dtype  [str, np.dtype, or ExtensionDtype, optional] Default None.

Returns

ExtensionArray or np.ndarray  A view on the ExtensionArray’s data.

pandas.api.extensions.ExtensionArray._concat_same_type

classmethod ExtensionArray._concat_same_type(to_concat)
Concatenate multiple array of this dtype.

Parameters

to_concat  [sequence of this type]

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray._formatter

ExtensionArray._formatter(boxed=False)
Formatting function for scalar values.
This is used in the default ‘__repr__’. The returned formatting function receives instances of your scalar type.

Parameters

boxed  [bool, default False] An indicated for whether or not your array is being printed within a Series, DataFrame, or Index (True), or just by itself (False). This may be useful if you want scalar values to appear differently within a Series versus on its own (e.g. quoted or not).

Returns

Callable[[Any], str] A callable that gets instances of the scalar type and returns a string. By default, repr() is used when boxed=False and str() is used when boxed=True.
pandas.api.extensions.ExtensionArray._from_factorized

**classmethod** ExtensionArray._from_factorized(values, original)

Reconstruct an ExtensionArray after factorization.

**Parameters**

- **values** [ndarray] An integer ndarray with the factorized values.
- **original** [ExtensionArray] The original ExtensionArray that factorize was called on.

**See also:**

- factorize Top-level factorize method that dispatches here.
- ExtensionArray.factorize Encode the extension array as an enumerated type.

pandas.api.extensions.ExtensionArray._from_sequence

**classmethod** ExtensionArray._from_sequence(scalars, *, dtype=None, copy=False)

Construct a new ExtensionArray from a sequence of scalars.

**Parameters**

- **scalars** [Sequence] Each element will be an instance of the scalar type for this array, cls.dtype.type or be converted into this type in this method.
- **dtype** [dtype, optional] Construct for this particular dtype. This should be a Dtype compatible with the ExtensionArray.
- **copy** [bool, default False] If True, copy the underlying data.

**Returns**

ExtensionArray

pandas.api.extensions.ExtensionArray._from_sequence_of_strings

**classmethod** ExtensionArray._from_sequence_of_strings(strings, *, dtype=None, copy=False)

Construct a new ExtensionArray from a sequence of strings.

**Parameters**

- **strings** [Sequence] Each element will be an instance of the scalar type for this array, cls.dtype.type.
- **dtype** [dtype, optional] Construct for this particular dtype. This should be a Dtype compatible with the ExtensionArray.
- **copy** [bool, default False] If True, copy the underlying data.

**Returns**

ExtensionArray

3.15. Extensions 2777
pandas.api.extensions.ExtensionArray._reduce

ExtensionArray._reduce(name, *, skipna=True, **kwargs)

Return a scalar result of performing the reduction operation.

Parameters

- **name** [str] Name of the function, supported values are: { any, all, min, max, sum, mean, median, prod, std, var, sem, kurt, skew }.
- **skipna** [bool, default True] If True, skip NaN values.
- **kwargs** Additional keyword arguments passed to the reduction function. Currently, ddof is the only supported kwarg.

Returns

- **scalar**

Raises

- **TypeError** [subclass does not define reductions]

pandas.api.extensions.ExtensionArray._values_for_argsort

ExtensionArray._values_for_argsort()

Return values for sorting.

Returns

- **ndarray** The transformed values should maintain the ordering between values within the array.

See also:

- **ExtensionArray.argsort** Return the indices that would sort this array.

pandas.api.extensions.ExtensionArray._values_for_factorize

ExtensionArray._values_for_factorize()

Return an array and missing value suitable for factorization.

Returns

- **values** [ndarray] An array suitable for factorization. This should maintain order and be a supported dtype (Float64, Int64, UInt64, String, Object). By default, the extension array is cast to object dtype.
- **na_value** [object] The value in values to consider missing. This will be treated as NA in the factorization routines, so it will be coded as na_sentinel and not included in uniques. By default, np.nan is used.
3.15.7 pandas.arrays.PandasArray

class pandas.arrays.PandasArray(values, copy=False)
A pandas ExtensionArray for NumPy data.

This is mostly for internal compatibility, and is not especially useful on its own.

Parameters

values [ndarray] The NumPy ndarray to wrap. Must be 1-dimensional.

copy [bool, default False] Whether to copy values.

Attributes

None

Methods

None

Additionally, we have some utility methods for ensuring your object behaves correctly.

3.15.8 pandas.api.indexers.check_array_indexer

pandas.api.indexers.check_array_indexer(array, indexer)
Check if indexer is a valid array indexer for array.

For a boolean mask, array and indexer are checked to have the same length. The dtype is validated, and if it is an integer or boolean ExtensionArray, it is checked if there are missing values present, and it is converted to the appropriate numpy array. Other dtypes will raise an error.

Non-array indexers (integer, slice, Ellipsis, tuples, ..) are passed through as is.

New in version 1.0.0.

Parameters

array [array-like] The array that is being indexed (only used for the length).

indexer [array-like or list-like] The array-like that’s used to index. List-like input that is not yet a numpy array or an ExtensionArray is converted to one. Other input types are passed through as is.

Returns

numppy.ndarray The validated indexer as a numpy array that can be used to index.
**Raises**

- **IndexError** When the lengths don’t match.
- **ValueError** When *indexer* cannot be converted to a numpy ndarray to index (e.g. presence of missing values).

**See also:**

- `api.types.is_bool_dtype` Check if *key* is of boolean dtype.

**Examples**

When checking a boolean mask, a boolean ndarray is returned when the arguments are all valid.

```python
>>> mask = pd.array([True, False])
>>> arr = pd.array([1, 2])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

An IndexError is raised when the lengths don’t match.

```python
>>> mask = pd.array([True, False, True])
>>> pd.api.indexers.check_array_indexer(arr, mask)
Traceback (most recent call last):
 ...
 IndexError: Boolean index has wrong length: 3 instead of 2.
```

NA values in a boolean array are treated as False.

```python
>>> mask = pd.array([True, pd.NA])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

A numpy boolean mask will get passed through (if the length is correct):

```python
>>> mask = np.array([True, False])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

Similarly for integer indexers, an integer ndarray is returned when it is a valid indexer, otherwise an error is (for integer indexers, a matching length is not required):

```python
>>> indexer = pd.array([0, 2], dtype="Int64")
>>> arr = pd.array([1, 2, 3])
>>> pd.api.indexers.check_array_indexer(arr, indexer)
array([0, 2])
```

```python
>>> indexer = pd.array([0, pd.NA], dtype="Int64")
>>> pd.api.indexers.check_array_indexer(arr, indexer)
Traceback (most recent call last):
 ...
 ValueError: Cannot index with an integer indexer containing NA values
```

For non-integer/boolean dtypes, an appropriate error is raised:
>>> indexer = np.array([0., 2.], dtype="float64")
>>> pd.api.indexers.check_array_indexer(arr, indexer)
Traceback (most recent call last):
...
IndexError: arrays used as indices must be of integer or boolean type

The sentinel pandas.api.extensions.no_default is used as the default value in some methods. Use an is comparison to check if the user provides a non-default value.
4.1 Contributing to pandas

Table of contents:

• Where to start?
• Bug reports and enhancement requests
• Working with the code
  – Version control, Git, and GitHub
  – Getting started with Git
  – Forking
  – Creating a branch
• Contributing your changes to pandas
  – Committing your code
  – Pushing your changes
  – Review your code
  – Finally, make the pull request
  – Updating your pull request
  – Autofixing formatting errors
  – Delete your merged branch (optional)
• Tips for a successful pull request
4.1.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are brand new to pandas or open-source development, we recommend going through the GitHub “issues” tab to find issues that interest you. There are a number of issues listed under Docs and good first issue where you could start out. Once you’ve found an interesting issue, you can return here to get your development environment setup.

When you start working on an issue, it’s a good idea to assign the issue to yourself, so nobody else duplicates the work on it. GitHub restricts assigning issues to maintainers of the project only. In most projects, and until recently in pandas, contributors added a comment letting others know they are working on an issue. While this is ok, you need to check each issue individually, and it’s not possible to find the unassigned ones.

For this reason, we implemented a workaround consisting of adding a comment with the exact text `take`. When you do it, a GitHub action will automatically assign you the issue (this will take seconds, and may require refreshing the page to see it). By doing this, it’s possible to filter the list of issues and find only the unassigned ones.

So, a good way to find an issue to start contributing to pandas is to check the list of unassigned good first issues and assign yourself one you like by writing a comment with the exact text `take`.

If for whatever reason you are not able to continue working with the issue, please try to unassign it, so other people know it’s available again. You can check the list of assigned issues, since people may not be working in them anymore. If you want to work on one that is assigned, feel free to kindly ask the current assignee if you can take it (please allow at least a week of inactivity before considering work in the issue discontinued).

Feel free to ask questions on the mailing list or on Gitter.

4.1.2 Bug reports and enhancement requests

Bug reports are an important part of making pandas more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing. See this stackoverflow article and this blogpost for tips on writing a good bug report.

Trying the bug-producing code out on the main branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self-contained Python snippet reproducing the problem. You can format the code nicely by using GitHub Flavored Markdown:

   ```python
   >>> from pandas import DataFrame
   >>> df = DataFrame(...)  
   ```

2. Include the full version string of pandas and its dependencies. You can use the built-in function:

   ```python
   >>> import pandas as pd
   >>> pd.show_versions()
   ```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the pandas community and be open to comments/ideas from others.
4.1.3 Working with the code

Now that you have an issue you want to fix, enhancement to add, or documentation to improve, you need to learn how to work with GitHub and the pandas code base.

Version control, Git, and GitHub

To the new user, working with Git is one of the more daunting aspects of contributing to pandas. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:
- the GitHub help pages.
- the NumPy documentation.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the pandas project page and hit the Fork button. You will want to clone your fork to your machine:

```bash
git clone https://github.com/your-user-name/pandas.git pandas-yourname
cd pandas-yourname
git remote add upstream https://github.com/pandas-dev/pandas.git
```

This creates the directory pandas-yourname and connects your repository to the upstream (main project) pandas repository.

Note that performing a shallow clone (with `--depth=N`, for some N greater or equal to 1) might break some tests and features as `pd.show_versions()` as the version number cannot be computed anymore.

Creating a branch

You want your main branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```bash
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```bash
git checkout -b shiny-new-feature
```
This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to pandas. You can have many shiny-new-features and switch in between them using the git checkout command.

When creating this branch, make sure your main branch is up to date with the latest upstream main version. To update your local main branch, you can do:

```
git checkout main
git pull upstream main --ff-only
```

When you want to update the feature branch with changes in main after you created the branch, check the section on updating a PR.

4.1.4 Contributing your changes to pandas

Committing your code

Keep style fixes to a separate commit to make your pull request more readable.

Once you’ve made changes, you can see them by typing:

```
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```
git add path/to/file-to-be-added.py
```

Doing ‘git status’ again should give something like:

```
# On branch shiny-new-feature
#
#    modified: /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message. pandas uses a convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- **ENH**: Enhancement, new functionality
- **BUG**: Bug fix
- **DOC**: Additions/updates to documentation
- **TST**: Additions/updates to tests
- **BLD**: Updates to the build process/scripts
- **PERF**: Performance improvement
- **TYP**: Type annotations
- **CLN**: Code cleanup

The following defines how a commit message should be structured. Please reference the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < 80 chars.
- One blank line.
• Optionally, a commit message body.

Now you can commit your changes in your local repository:

```
git commit -m
```

**Pushing your changes**

When you want your changes to appear publicly on your GitHub page, push your forked feature branch’s commits:

```
git push origin shiny-new-feature
```

Here `origin` is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin git@github.com:yourname/pandas.git (fetch)
origin git@github.com:yourname/pandas.git (push)
upstream git://github.com/pandas-dev/pandas.git (fetch)
upstream git://github.com/pandas-dev/pandas.git (push)
```

Now your code is on GitHub, but it is not yet a part of the pandas project. For that to happen, a pull request needs to be submitted on GitHub.

**Review your code**

When you’re ready to ask for a code review, file a pull request. Before you do, once again make sure that you have followed all the guidelines outlined in this document regarding code style, tests, performance tests, and documentation. You should also double check your branch changes against the branch it was based on:

1. Navigate to your repository on GitHub – https://github.com/your-user-name/pandas
2. Click on **Branches**
3. Click on the **Compare** button for your feature branch
4. Select the **base** and **compare** branches, if necessary. This will be **main** and **shiny-new-feature**, respectively.

**Finally, make the pull request**

If everything looks good, you are ready to make a pull request. A pull request is how code from a local repository becomes available to the GitHub community and can be looked at and eventually merged into the main version. This pull request and its associated changes will eventually be committed to the main branch and available in the next release. To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the **Pull Request** button
3. You can then click on **Commits** and **Files Changed** to make sure everything looks okay one last time
4. Write a description of your changes in the **Preview Discussion** tab
5. Click **Send Pull Request**.

This request then goes to the repository maintainers, and they will review the code.
Updating your pull request

Based on the review you get on your pull request, you will probably need to make some changes to the code. In that case, you can make them in your branch, add a new commit to that branch, push it to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the Continuous Integration tests.

Another reason you might need to update your pull request is to solve conflicts with changes that have been merged into the main branch since you opened your pull request.

To do this, you need to “merge upstream main” in your branch:

```
git checkout shiny-new-feature
git fetch upstream
git merge upstream/main
```

If there are no conflicts (or they could be fixed automatically), a file with a default commit message will open, and you can simply save and quit this file.

If there are merge conflicts, you need to solve those conflicts. See for example at https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/ for an explanation on how to do this. Once the conflicts are merged and the files where the conflicts were solved are added, you can run `git commit` to save those fixes.

If you have uncommitted changes at the moment you want to update the branch with main, you will need to `stash` them prior to updating (see the `stash docs`). This will effectively store your changes and they can be reapplied after updating.

After the feature branch has been update locally, you can now update your pull request by pushing to the branch on GitHub:

```
git push origin shiny-new-feature
```

Autofixing formatting errors

We use several styling checks (e.g. black, flake8, isort) which are run after you make a pull request. If there is a scenario where any of these checks fail then you can comment:

```
@github-actions pre-commit
```

on that pull request. This will trigger a workflow which will autofix formatting errors.

To automatically fix formatting errors on each commit you make, you can set up pre-commit yourself. First, create a Python environment and then set up pre-commit.

Delete your merged branch (optional)

Once your feature branch is accepted into upstream, you’ll probably want to get rid of the branch. First, merge upstream main into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout main
git merge upstream/main
```

Then you can do:
git branch -d shiny-new-feature

Make sure you use a lower-case -d, or else git won’t warn you if your feature branch has not actually been merged.
The branch will still exist on GitHub, so to delete it there do:

```bash
git push origin --delete shiny-new-feature
```

### 4.1.5 Tips for a successful pull request

If you have made it to the *Review your code* phase, one of the core contributors may take a look. Please note however that a handful of people are responsible for reviewing all of the contributions, which can often lead to bottlenecks.

To improve the chances of your pull request being reviewed, you should:

- **Reference an open issue** for non-trivial changes to clarify the PR’s purpose
- **Ensure you have appropriate tests**. These should be the first part of any PR
- **Keep your pull requests as simple as possible**. Larger PRs take longer to review
- **Ensure that CI is in a green state**. Reviewers may not even look otherwise
- **Keep Updating your pull request**, either by request or every few days

### 4.2 Creating a development environment

To test out code changes, you’ll need to build pandas from source, which requires a C/C++ compiler and Python environment. If you’re making documentation changes, you can skip to *contributing to the documentation* but if you skip creating the development environment you won’t be able to build the documentation locally before pushing your changes.

#### Table of contents:

- Creating an environment using Docker
  - Installing a C compiler
  - Creating a Python environment
  - Creating a Python environment (pip)

#### 4.2.1 Creating an environment using Docker

Instead of manually setting up a development environment, you can use Docker to automatically create the environment with just several commands. pandas provides a Dockerfile in the root directory to build a Docker image with a full pandas development environment.

**Docker Commands**

Build the Docker image:
# Build the image pandas-yourname-env

docker build --tag pandas-yourname-env .

# Or build the image by passing your GitHub username to use your own fork

docker build --build-arg gh_username=yourname --tag pandas-yourname-env .

Run Container:

# Run a container and bind your local repo to the container

docker run -it -w /home/pandas --rm -v path-to-local-pandas-repo:/home/pandas pandas-yourname-env

Note: If you bind your local repo for the first time, you have to build the C extensions afterwards. Run the following command inside the container:

```bash
python setup.py build_ext -j 4
```

You need to rebuild the C extensions anytime the Cython code in pandas/_libs changes. This most frequently occurs when changing or merging branches.

Even easier, you can integrate Docker with the following IDEs:

**Visual Studio Code**

You can use the DockerFile to launch a remote session with Visual Studio Code, a popular free IDE, using the .devcontainer.json file. See [https://code.visualstudio.com/docs/remote/containers](https://code.visualstudio.com/docs/remote/containers) for details.

**PyCharm (Professional)**

Enable Docker support and use the Services tool window to build and manage images as well as run and interact with containers. See [https://www.jetbrains.com/help/pycharm/docker.html](https://www.jetbrains.com/help/pycharm/docker.html) for details.

### 4.2.2 Creating an environment without Docker

**Installing a C compiler**

pandas uses C extensions (mostly written using Cython) to speed up certain operations. To install pandas from source, you need to compile these C extensions, which means you need a C compiler. This process depends on which platform you’re using.

If you have setup your environment using conda, the packages c-compiler and cxx-compiler will install a fitting compiler for your platform that is compatible with the remaining conda packages. On Windows and macOS, you will also need to install the SDKs as they have to be distributed separately. These packages will automatically be installed by using the `pandas environment.yml` file.

**Windows**

You will need **Build Tools for Visual Studio 2019**.


You can install the necessary components on the commandline using `vs_buildtools.exe`:
vs_buildertools.exe --quiet --wait --norestart --nocache ^
--installPath C:\BuildTools ^
--add "Microsoft.VisualStudio.Workload.VCTools;includeRecommended" ^
--add Microsoft.VisualStudio.Component.VC.v141 ^
--add Microsoft.VisualStudio.Component.VC.v141.x86.x64 ^

to setup the right paths on the commandline, call "C:\BuildTools\VC\Auxiliary\Build\vcvars64.bat" ^
vcc_registration_ver=14.16 10.0.17763.0.

macOS

to use the conda-based compilers, you will need to install the Developer Tools using xcode-select --install.
otherwise information about compiler installation can be found here: https://devguide.python.org/setup/#macos

Linux

for Linux-based conda installations, you won’t have to install any additional components outside of the conda environment. the instructions below are only needed if your setup isn’t based on conda environments.

Some Linux distributions will come with a pre-installed C compiler. To find out which compilers (and versions) are installed on your system:

```
# for Debian/Ubuntu:
dpkg --list | grep compiler
# for Red Hat/RHEL/CentOS/Fedora:
yum list installed | grep -i --color compiler
```

GCC (GNU Compiler Collection), is a widely used compiler, which supports C and a number of other languages. if GCC is listed as an installed compiler nothing more is required. if no C compiler is installed (or you wish to install a newer version) you can install a compiler (GCC in the example code below) with:

```
# for recent Debian/Ubuntu:
sudo apt install build-essential
# for Red Hat/RHEL/CentOS/Fedora
yum groupinstall "Development Tools"
```

for other Linux distributions, consult your favorite search engine for compiler installation instructions.

let us know if you have any difficulties by opening an issue or reaching out on Gitter.

Creating a Python environment

Now create an isolated pandas development environment:

- Install either Anaconda, miniconda, or miniforge
- Make sure your conda is up to date (conda update conda)
- Make sure that you have cloned the repository
- cd to the pandas source directory

We’ll now kick off a three-step process:

1. Install the build dependencies
2. Build and install pandas
3. Install the optional dependencies

4.2. Creating a development environment 2791
Create and activate the build environment
conda env create -f environment.yml
conda activate pandas-dev

# or with older versions of Anaconda:
source activate pandas-dev

# Build and install pandas
python setup.py build_ext -j 4
python -m pip install -e . --no-build-isolation --no-use-pep517

At this point you should be able to import pandas from your locally built version:

```
$ python
>>> import pandas
>>> print(pandas.__version__)
0.22.0.dev0+29.g4ad6d4d74
```

This will create the new environment, and not touch any of your existing environments, nor any existing Python installation.

To view your environments:

```
conda info -e
```

To return to your root environment:

```
conda deactivate
```

See the full conda docs here.

**Creating a Python environment (pip)**

If you aren’t using conda for your development environment, follow these instructions. You’ll need to have at least the *minimum Python version* that pandas supports. You also need to have *setuptools* 51.0.0 or later to build pandas.

**Unix/macOS with virtualenv**

```
# Create a virtual environment
# Use an ENV_DIR of your choice. We’ll use ~/virtualenvs/pandas-dev
# Any parent directories should already exist
python3 -m venv ~/virtualenvs/pandas-dev

# Activate the virtualenv
. ~/virtualenvs/pandas-dev/bin/activate

# Install the build dependencies
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext -j 4
python -m pip install -e . --no-build-isolation --no-use-pep517
```

**Unix/macOS with pyenv**
Consult the docs for setting up pyenv here.

```
# Create a virtual environment
# Use an ENV_DIR of your choice. We'll use ~/Users/<yourname>/pyenv/versions/pandas-dev
pyenv virtualenv <version> <name-to-give-it>

# For instance:
pyenv virtualenv 3.7.6 pandas-dev

# Activate the virtualenv
pyenv activate pandas-dev

# Now install the build dependencies in the cloned pandas repo
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext -j 4
python -m pip install -e . --no-build-isolation --no-use-pep517
```

**Windows**

Below is a brief overview on how to set-up a virtual environment with Powershell under Windows. For details please refer to the official virtualenv user guide

Use an ENV_DIR of your choice. We’ll use ~\virtualenvs\pandas-dev where ‘~’ is the folder pointed to by either $env:USERPROFILE (Powershell) or %USERPROFILE% (cmd.exe) environment variable. Any parent directories should already exist.

```
# Create a virtual environment
python -m venv $env:USERPROFILE\virtualenvs\pandas-dev

# Activate the virtualenv. Use activate.bat for cmd.exe
~\virtualenvs\pandas-dev\Scripts\Activate.ps1

# Install the build dependencies
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext -j 4
python -m pip install -e . --no-build-isolation --no-use-pep517
```

### 4.3 Contributing to the documentation

Contributing to the documentation benefits everyone who uses pandas. We encourage you to help us improve the documentation, and you don’t have to be an expert on pandas to do so! In fact, there are sections of the docs that are worse off after being written by experts. If something in the docs doesn’t make sense to you, updating the relevant section after you figure it out is a great way to ensure it will help the next person.

**Documentation:**

- About the pandas documentation
4.3.1 About the pandas documentation

The documentation is written in reStructuredText, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The pandas documentation consists of two parts: the docstrings in the code itself and the docs in this folder doc/.
  
The docstrings provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what’s new, installation, etc).

- The docstrings follow a pandas convention, based on the Numpy Docstring Standard. Follow the pandas docstring guide for detailed instructions on how to write a correct docstring.

pandas docstring guide

About docstrings and standards

A Python docstring is a string used to document a Python module, class, function or method, so programmers can understand what it does without having to read the details of the implementation.

Also, it is a common practice to generate online (html) documentation automatically from docstrings. Sphinx serves this purpose.

The next example gives an idea of what a docstring looks like:

```python
def add(num1, num2):
    r""
    Add up two integer numbers.
    This function simply wraps the `+` operator, and does not do anything interesting, except for illustrating what the docstring of a very simple function looks like.
    Parameters
    ----------
    num1 : int
        First number to add.
    num2 : int
        Second number to add.
    """
```

(continues on next page)
Some standards regarding docstrings exist, which make them easier to read, and allow them to be easily exported to other formats such as html or pdf.

The first conventions every Python docstring should follow are defined in PEP-257.

As PEP-257 is quite broad, other more specific standards also exist. In the case of pandas, the NumPy docstring convention is followed. These conventions are explained in this document:

- numpydoc docstring guide (which is based in the original Guide to NumPy/SciPy documentation)

numpydoc is a Sphinx extension to support the NumPy docstring convention.

The standard uses reStructuredText (reST). reStructuredText is a markup language that allows encoding styles in plain text files. Documentation about reStructuredText can be found in:

- Sphinx reStructuredText primer
- Quick reStructuredText reference
- Full reStructuredText specification

pandas has some helpers for sharing docstrings between related classes, see Sharing docstrings.

The rest of this document will summarize all the above guidelines, and will provide additional conventions specific to the pandas project.

## Writing a docstring

### General rules

Docstrings must be defined with three double-quotes. No blank lines should be left before or after the docstring. The text starts in the next line after the opening quotes. The closing quotes have their own line (meaning that they are not at the end of the last sentence).

On rare occasions reST styles like bold text or italics will be used in docstrings, but it is common to have inline code, which is presented between backticks. The following are considered inline code:
-- The name of a parameter
-- Python code, a module, function, built-in, type, literal... (e.g. os, list, numpy.abs, datetime.date, True)
-- A pandas class (in the form `:class:`pandas.Series`)
-- A pandas method (in the form `:meth:`pandas.Series.sum`)
-- A pandas function (in the form `:func:`pandas.to_datetime`)

**Note:** To display only the last component of the linked class, method or function, prefix it with ~. For example, `:class:`~pandas.Series` will link to pandas.Series but only display the last part, Series as the link text. See Sphinx cross-referencing syntax for details.

Good:

```python
def add_values(arr):
    """
    Add the values in `arr`.
    This is equivalent to Python `sum` of `meth:pandas.Series.sum`.
    Some sections are omitted here for simplicity.
    """
    return sum(arr)
```

Bad:

```python
def func():
    """
    Some function.
    With several mistakes in the docstring.
    It has a blank like after the signature `def func:`.
    The text 'Some function' should go in the line after the opening quotes of the docstring, not in the same line.
    There is a blank line between the docstring and the first line of code `foo = 1`.
    The closing quotes should be in the next line, not in this one.""
    foo = 1
    bar = 2
    return foo + bar
```
**Section 1: short summary**

The short summary is a single sentence that expresses what the function does in a concise way.

The short summary must start with a capital letter, end with a dot, and fit in a single line. It needs to express what the object does without providing details. For functions and methods, the short summary must start with an infinitive verb.

**Good:**

```python
def astype(dtype):
    """
    Cast Series type.
    This section will provide further details.
    """
    pass
```

**Bad:**

```python
def astype(dtype):
    """
    Casts Series type.
    Verb in third-person of the present simple, should be infinitive.
    """
    pass
```

```python
def astype(dtype):
    """
    Method to cast Series type.
    Does not start with verb.
    """
    pass
```

```python
def astype(dtype):
    """
    Cast Series type
    Missing dot at the end.
    """
    pass
```

```python
def astype(dtype):
    """
    Cast Series type from its current type to the new type defined in the parameter dtype.
    Summary is too verbose and doesn't fit in a single line.
    """
    pass
```
Section 2: extended summary

The extended summary provides details on what the function does. It should not go into the details of the parameters, or discuss implementation notes, which go in other sections.

A blank line is left between the short summary and the extended summary. Every paragraph in the extended summary ends with a dot.

The extended summary should provide details on why the function is useful and their use cases, if it is not too generic.

```python
def unstack():
    """
    Pivot a row index to columns.

    When using a MultiIndex, a level can be pivoted so each value in
    the index becomes a column. This is especially useful when a subindex
    is repeated for the main index, and data is easier to visualize as a
    pivot table.

    The index level will be automatically removed from the index when added
    as columns.
    """
    pass
```

Section 3: parameters

The details of the parameters will be added in this section. This section has the title “Parameters”, followed by a line with a hyphen under each letter of the word “Parameters”. A blank line is left before the section title, but not after, and not between the line with the word “Parameters” and the one with the hyphens.

After the title, each parameter in the signature must be documented, including *args and **kwargs, but not self.

The parameters are defined by their name, followed by a space, a colon, another space, and the type (or types). Note that the space between the name and the colon is important. Types are not defined for *args and **kwargs, but must be defined for all other parameters. After the parameter definition, it is required to have a line with the parameter description, which is indented, and can have multiple lines. The description must start with a capital letter, and finish with a dot.

For keyword arguments with a default value, the default will be listed after a comma at the end of the type. The exact form of the type in this case will be “int, default 0”. In some cases it may be useful to explain what the default argument means, which can be added after a comma “int, default -1, meaning all cpus”.

In cases where the default value is None, meaning that the value will not be used. Instead of "str, default None", it is preferred to write "str, optional". When None is a value being used, we will keep the form "str, default None". For example, in df.to_csv(compression=None), None is not a value being used, but means that compression is optional, and no compression is being used if not provided. In this case we will use "str, optional". Only in cases like func(value=None) and None is being used in the same way as 0 or foo would be used, then we will specify “str, int or None, default None”.

Good:

```python
class Series:
    def plot(self, kind, color='blue', **kwargs):
```

(continues on next page)
"""
Generate a plot.

Render the data in the Series as a matplotlib plot of the specified kind.

Parameters
----------
kind : str
    Kind of matplotlib plot.
color : str, default 'blue'
    Color name or rgb code.
**kwargs
    These parameters will be passed to the matplotlib plotting function.
"""

pass

Bad:

class Series:
    def plot(self, kind, **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the specified kind.

        Note the blank line between the parameters title and the first parameter. Also, note that after the name of the parameter `kind` and before the colon, a space is missing.

        Also, note that the parameter descriptions do not start with a capital letter, and do not finish with a dot.

        Finally, the `**kwargs` parameter is missing.

        Parameters
        ----------

        kind: str
            kind of matplotlib plot
        """

        pass
Parameter types

When specifying the parameter types, Python built-in data types can be used directly (the Python type is preferred to the more verbose string, integer, boolean, etc):

- int
- float
- str
- bool

For complex types, define the subtypes. For dict and tuple, as more than one type is present, we use the brackets to help read the type (curly brackets for dict and normal brackets for tuple):

- list of int
- dict of {str : int}
- tuple of (str, int, int)
- tuple of (str,)
- set of str

In case where there are just a set of values allowed, list them in curly brackets and separated by commas (followed by a space). If the values are ordinal and they have an order, list them in this order. Otherwise, list the default value first, if there is one:

- {0, 10, 25}
- {'simple', 'advanced'}
- {'low', 'medium', 'high'}
- {'cat', 'dog', 'bird'}

If the type is defined in a Python module, the module must be specified:

- datetime.date
- datetime.datetime
- decimal.Decimal

If the type is in a package, the module must be also specified:

- numpy.ndarray
- scipy.sparse.coo_matrix

If the type is a pandas type, also specify pandas except for Series and DataFrame:

- Series
- DataFrame
- pandas.Index
- pandas.Categorical
- pandas.arrays.SparseArray

If the exact type is not relevant, but must be compatible with a NumPy array, array-like can be specified. If Any type that can be iterated is accepted, iterable can be used:

- array-like
If more than one type is accepted, separate them by commas, except the last two types, that need to be separated by the word ‘or’:

- int or float
- float, decimal.Decimal or None
- str or list of str

If None is one of the accepted values, it always needs to be the last in the list.

For axis, the convention is to use something like:

- axis : {0 or ‘index’, 1 or ‘columns’, None}, default None

### Section 4: returns or yields

If the method returns a value, it will be documented in this section. Also if the method yields its output.

The title of the section will be defined in the same way as the “Parameters”. With the names “Returns” or “Yields” followed by a line with as many hyphens as the letters in the preceding word.

The documentation of the return is also similar to the parameters. But in this case, no name will be provided, unless the method returns or yields more than one value (a tuple of values).

The types for “Returns” and “Yields” are the same as the ones for the “Parameters”. Also, the description must finish with a dot.

For example, with a single value:

```python
def sample():
    """
    Generate and return a random number.
    The value is sampled from a continuous uniform distribution between 0 and 1.
    """
    return np.random.random()
```

With more than one value:

```python
import string

def random_letters():
    """
    Generate and return a sequence of random letters.
    The length of the returned string is also random, and is also returned.
    """
    return np.random.random()
```

(continues on next page)
-------
length : int
    Length of the returned string.
letters : str
    String of random letters.

```
length = np.random.randint(1, 10)
letters = ''.join(np.random.choice(string.ascii_lowercase)
                 for i in range(length))
return length, letters
```

If the method yields its value:

```python
def sample_values():
    """
    Generate an infinite sequence of random numbers.

    The values are sampled from a continuous uniform distribution between
    0 and 1.
    """

    while True:
        yield np.random.random()
```

Section 5: see also

This section is used to let users know about pandas functionality related to the one being documented. In rare cases, if no related methods or functions can be found at all, this section can be skipped.

An obvious example would be the head() and tail() methods. As tail() does the equivalent as head() but at the end of the Series or DataFrame instead of at the beginning, it is good to let the users know about it.

To give an intuition on what can be considered related, here are some examples:

- loc and iloc, as they do the same, but in one case providing indices and in the other positions
- max and min, as they do the opposite
- iterrows, itertuples and items, as it is easy that a user looking for the method to iterate over columns ends up in the method to iterate over rows, and vice-versa
- fillna and dropna, as both methods are used to handle missing values
- read_csv and to_csv, as they are complementary
- merge and join, as one is a generalization of the other
- astype and pandas.to_datetime, as users may be reading the documentation of astype to know how to cast as a date, and the way to do it is with pandas.to_datetime
- where is related to numpy.where, as its functionality is based on it
When deciding what is related, you should mainly use your common sense and think about what can be useful for the users reading the documentation, especially the less experienced ones.

When relating to other libraries (mainly numpy), use the name of the module first (not an alias like np). If the function is in a module which is not the main one, like scipy.sparse, list the full module (e.g. scipy.sparse.coo_matrix).

This section has a header, “See Also” (note the capital S and A), followed by the line with hyphens and preceded by a blank line.

After the header, we will add a line for each related method or function, followed by a space, a colon, another space, and a short description that illustrates what this method or function does, why it is relevant in this context, and what the key differences are between the documented function and the one being referenced. The description must also end with a dot.

Note that in “Returns” and “Yields”, the description is located on the line after the type. In this section, however, it is located on the same line, with a colon in between. If the description does not fit on the same line, it can continue onto other lines which must be further indented.

For example:

```python
class Series:
    def head(self):
        """
        Return the first 5 elements of the Series.

        This function is mainly useful to preview the values of the Series without displaying the whole of it.
        """
        Returns
        -------
        Series
        Subset of the original series with the 5 first values.

        See Also
        --------
        Series.tail : Return the last 5 elements of the Series.
        Series.iloc : Return a slice of the elements in the Series, which can also be used to return the first or last n.
        """
        return self.iloc[:5]
```

Section 6: notes

This is an optional section used for notes about the implementation of the algorithm, or to document technical aspects of the function behavior.

Feel free to skip it, unless you are familiar with the implementation of the algorithm, or you discover some counter-intuitive behavior while writing the examples for the function.

This section follows the same format as the extended summary section.
Section 7: examples

This is one of the most important sections of a docstring, despite being placed in the last position, as often people understand concepts better by example than through accurate explanations.

Examples in docstrings, besides illustrating the usage of the function or method, must be valid Python code, that returns the given output in a deterministic way, and that can be copied and run by users.

Examples are presented as a session in the Python terminal. >>> is used to present code. ... is used for code continuing from the previous line. Output is presented immediately after the last line of code generating the output (no blank lines in between). Comments describing the examples can be added with blank lines before and after them.

The way to present examples is as follows:

1. Import required libraries (except numpy and pandas)
2. Create the data required for the example
3. Show a very basic example that gives an idea of the most common use case
4. Add examples with explanations that illustrate how the parameters can be used for extended functionality

A simple example could be:

```python
class Series:

    def head(self, n=5):
        """
        Return the first elements of the Series.

        This function is mainly useful to preview the values of the Series without displaying all of it.

        Parameters
        ----------
        n : int
            Number of values to return.

        Return
        ------
        pandas.Series
            Subset of the original series with the n first values.

        See Also
        --------
        tail : Return the last n elements of the Series.

        Examples
        --------
        >>> s = pd.Series(['Ant', 'Bear', 'Cow', 'Dog', 'Falcon',
        ...                 'Lion', 'Monkey', 'Rabbit', 'Zebra'])
        >>> s.head()
        0   Ant
        1   Bear
        2   Cow
        3   Dog
```

(continues on next page)
4 Falcon
dtype: object

With the `n` parameter, we can change the number of returned rows:

```python
>>> s.head(n=3)
0  Ant
1  Bear
2  Cow
dtype: object

return self.iloc[:n]
```

The examples should be as concise as possible. In cases where the complexity of the function requires long examples, it is recommended to use blocks with headers in bold. Use double star ** to make a text bold, like in **this example**.

**Conventions for the examples**

Code in examples is assumed to always start with these two lines which are not shown:

```python
import numpy as np
import pandas as pd
```

Any other module used in the examples must be explicitly imported, one per line (as recommended in PEP 8#imports) and avoiding aliases. Avoid excessive imports, but if needed, imports from the standard library go first, followed by third-party libraries (like matplotlib).

When illustrating examples with a single `Series` use the name `s`, and if illustrating with a single `DataFrame` use the name `df`. For indices, `idx` is the preferred name. If a set of homogeneous `Series` or `DataFrame` is used, name them `s1`, `s2`, `s3`... or `df1`, `df2`, `df3`... If the data is not homogeneous, and more than one structure is needed, name them with something meaningful, for example `df_main` and `df_to_join`.

Data used in the example should be as compact as possible. The number of rows is recommended to be around 4, but make it a number that makes sense for the specific example. For example in the head method, it requires to be higher than 5, to show the example with the default values. If doing the `mean`, we could use something like `[1, 2, 3]`, so it is easy to see that the value returned is the mean.

For more complex examples (grouping for example), avoid using data without interpretation, like a matrix of random numbers with columns A, B, C, D... And instead use a meaningful example, which makes it easier to understand the concept. Unless required by the example, use names of animals, to keep examples consistent. And numerical properties of them.

When calling the method, keywords arguments `head(n=3)` are preferred to positional arguments `head(3)`.

**Good:**

```python
class Series:

    def mean(self):
        """
        Compute the mean of the input.
        """
        Examples

(continues on next page)
---
>>> s = pd.Series([1, 2, 3])
>>> s.mean()
2
pass

deffillna(self, value):
    """
Replace missing values by `value`.
    """

Examples
-------
>>> s = pd.Series([1, np.nan, 3])
>>> s.fillna(0)
[1, 0, 3]
pass


def groupby_mean(self):
    """
Group by index and return mean.
    """

Examples
-------
>>> s = pd.Series([380., 370., 24., 26],
... name='max_speed',
... index=['falcon', 'falcon', 'parrot', 'parrot'])
>>> s.groupby_mean()
index
falcon  375.0
parrot  25.0
Name: max_speed, dtype: float64
pass

def contains(self, pattern, case_sensitive=True, na=numpy.nan):
    """
Return whether each value contains `pattern`.

In this case, we are illustrating how to use sections, even
if the example is simple enough and does not require them.

Examples
-------
>>> s = pd.Series('Antelope', 'Lion', 'Zebras', np.nan)
>>> s.contains('A')
0  False
1  False
2  True
3  NaN
dtype: bool

**Case sensitivity**

With `case_sensitive` set to `False` we can match `a` with both `a` and `A`:

```python
>>> s.contains(pattern='a', case_sensitive=False)
0    True
1    False
2    True
3     NaN
dtype: bool
```

**Missing values**

We can fill missing values in the output using the `na` parameter:

```python
>>> s.contains(pattern='a', na=False)
0    False
1    False
2    True
3    False
dtype: bool
```

```
pass
```

**Bad:**

```python
def method(foo=None, bar=None):
    ""
    A sample DataFrame method.
    ""
    Do not import NumPy and pandas.
    Try to use meaningful data, when it makes the example easier to understand.
    Try to avoid positional arguments like in `df.method(1)`.
    They can be all right if previously defined with a meaningful name, like in `present_value(interest_rate)`,
    but avoid them otherwise.
    When presenting the behavior with different parameters, do not place all the calls one next to the other.
    Instead, add a short sentence explaining what the example shows.
    Examples
    --------
    >>> import numpy as np
    >>> import pandas as pd
    >>> df = pd.DataFrame(np.random.randn(3, 3),
    ... columns=('a', 'b', 'c'))
```

(continues on next page)
Tips for getting your examples pass the doctests

Getting the examples pass the doctests in the validation script can sometimes be tricky. Here are some attention points:

– Import all needed libraries (except for pandas and NumPy, those are already imported as `import pandas as pd` and `import numpy as np`) and define all variables you use in the example.

– Try to avoid using random data. However random data might be OK in some cases, like if the function you are documenting deals with probability distributions, or if the amount of data needed to make the function result meaningful is too much, such that creating it manually is very cumbersome. In those cases, always use a fixed random seed to make the generated examples predictable. Example:

```python
>>> np.random.seed(42)
>>> df = pd.DataFrame({'normal': np.random.normal(100, 5, 20)})
```

– If you have a code snippet that wraps multiple lines, you need to use `'...'` on the continued lines:

```python
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], index=['a', 'b', 'c'],
                    columns=['A', 'B'])
```

– If you want to show a case where an exception is raised, you can do:

```python
>>> pd.to_datetime(['712-01-01'])
Traceback (most recent call last):
  OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 712-01-01 00:00:00
```

It is essential to include the “Traceback (most recent call last):”", but for the actual error only the error name is sufficient.

– If there is a small part of the result that can vary (e.g. a hash in an object representation), you can use `...` to represent this part.

If you want to show that `s.plot()` returns a matplotlib AxesSubplot object, this will fail the doctest

```python
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at 0x7efd0c0b0690>
```

However, you can do (notice the comment that needs to be added)

```python
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at ...>
Plots in examples

There are some methods in pandas returning plots. To render the plots generated by the examples in the documentation, the .. plot:: directive exists.

To use it, place the next code after the “Examples” header as shown below. The plot will be generated automatically when building the documentation.

```python
class Series:
    def plot(self):
        """
        Generate a plot with the `Series` data.

        Examples
        --------
        .. plot::
            :context: close-figs
            >>> s = pd.Series([1, 2, 3])
            >>> s.plot()
        """
        pass
```

Sharing docstrings

pandas has a system for sharing docstrings, with slight variations, between classes. This helps us keep docstrings consistent, while keeping things clear for the user reading. It comes at the cost of some complexity when writing.

Each shared docstring will have a base template with variables, like `{klass}`. The variables filled in later on using the doc decorator. Finally, docstrings can also be appended to with the doc decorator.

In this example, we’ll create a parent docstring normally (this is like pandas.core.generic.NDFrame. Then we’ll have two children (like pandas.core.series.Series and pandas.core.frame.DataFrame). We’ll substitute the class names in this docstring.

```python
class Parent:
    @doc(klass="Parent")
    def my_function(self):
        """Apply my function to {klass}""
        ...

class ChildA(Parent):
    @doc(Parent.my_function, klass="ChildA")
    def my_function(self):
        ...

class ChildB(Parent):
    @doc(Parent.my_function, klass="ChildB")
    def my_function(self):
        ...
```

4.3. Contributing to the documentation 2809
The resulting docstrings are

```python
>>> print(Parent.my_function.__doc__)
Apply my function to Parent.
>>> print(ChildA.my_function.__doc__)
Apply my function to ChildA.
>>> print(ChildB.my_function.__doc__)
Apply my function to ChildB.
```

Notice:

1. We “append” the parent docstring to the children docstrings, which are initially empty.

Our files will often contain a module-level `_shared_doc_kwargs` with some common substitution values (things like `klass`, `axes`, etc).

You can substitute and append in one shot with something like

```python
@doc(template, **_shared_doc_kwargs)
def my_function(self):
    ...
```

where `template` may come from a module-level `_shared_docs` dictionary mapping function names to docstrings. Wherever possible, we prefer using `doc`, since the docstring-writing processes is slightly closer to normal.

See `pandas.core.generic.NDFrame.fillna` for an example template, and `pandas.core.series.Series.fillna` and `pandas.core.generic.frame.fillna` for the filled versions.

- The tutorials make heavy use of the IPython directive sphinx extension. This directive lets you put code in the documentation which will be run during the doc build. For example:

```ipython:: python
x = 2
x**3
```

will be rendered as:

```
In [1]: x = 2
In [2]: x**3
Out[2]: 8
```

Almost all code examples in the docs are run (and the output saved) during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

- Our API documentation files in `doc/source/reference` house the auto-generated documentation from the docstrings. For classes, there are a few subtleties around controlling which methods and attributes have pages auto-generated.

We have two autosummary templates for classes.

1. `_templates/autosummary/class.rst`. Use this when you want to automatically generate a page for every public method and attribute on the class. The Attributes and Methods sections will be automatically added to the class’ rendered documentation by numpydoc. See `DataFrame` for an example.

2. `_templates/autosummary/class_without_autosummary`. Use this when you want to pick a subset of methods / attributes to auto-generate pages for. When using this template, you should include an
Attributes and Methods section in the class docstring. See CategoricalIndex for an example.

Every method should be included in a toctree in one of the documentation files in doc/source/reference, else Sphinx will emit a warning.

**Note:** The .rst files are used to automatically generate Markdown and HTML versions of the docs. For this reason, please do not edit CONTRIBUTING.md directly, but instead make any changes to doc/source/development/contributing.rst. Then, to generate CONTRIBUTING.md, use pandoc with the following command:

```bash
pandoc doc/source/development/contributing.rst -t markdown_github > CONTRIBUTING.md
```

The utility script scripts/validate_docstrings.py can be used to get a csv summary of the API documentation. And also validate common errors in the docstring of a specific class, function or method. The summary also compares the list of methods documented in the files in doc/source/reference (which is used to generate the API Reference page) and the actual public methods. This will identify methods documented in doc/source/reference that are not actually class methods, and existing methods that are not documented in doc/source/reference.

### 4.3.2 Updating a pandas docstring

When improving a single function or method’s docstring, it is not necessarily needed to build the full documentation (see next section). However, there is a script that checks a docstring (for example for the DataFrame.mean method):

```bash
python scripts/validate_docstrings.py pandas.DataFrame.mean
```

This script will indicate some formatting errors if present, and will also run and test the examples included in the docstring. Check the pandas docstring guide for a detailed guide on how to format the docstring.

The examples in the docstring (‘doctests’) must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users. This can be checked with the script above, and is also tested on Travis. A failing doctest will be a blocker for merging a PR. Check the examples section in the docstring guide for some tips and tricks to get the doctests passing.

When doing a PR with a docstring update, it is good to post the output of the validation script in a comment on github.

### 4.3.3 How to build the pandas documentation

**Requirements**

First, you need to have a development environment to be able to build pandas (see the docs on creating a development environment).

**Building the documentation**

So how do you build the docs? Navigate to your local doc/ directory in the console and run:

```bash
python make.py html
```

Then you can find the HTML output in the folder doc/build/html/.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified. If you want to do a full clean build, do:
You can tell `make.py` to compile only a single section of the docs, greatly reducing the turn-around time for checking your changes.

```bash
# omit autosummary and API section
python make.py clean
python make.py --no-api

# compile the docs with only a single section, relative to the "source" folder.
# For example, compiling only this guide (doc/source/development/contributing.rst)
python make.py clean
python make.py --single development/contributing.rst

# compile the reference docs for a single function
python make.py clean
python make.py --single pandas.DataFrame.join

# compile whatsnew and API section (to resolve links in the whatsnew)
python make.py clean
python make.py --whatsnew
```

For comparison, a full documentation build may take 15 minutes, but a single section may take 15 seconds. Subsequent builds, which only process portions you have changed, will be faster.

The build will automatically use the number of cores available on your machine to speed up the documentation build. You can override this:

```bash
python make.py html --num-jobs 4
```

Open the following file in a web browser to see the full documentation you just built:

```bash
doc/build/html/index.html
```

And you'll have the satisfaction of seeing your new and improved documentation!

### Building main branch documentation

When pull requests are merged into the pandas main branch, the main parts of the documentation are also built by Travis-CI. These docs are then hosted [here](#), see also the Continuous Integration section.

#### 4.3.4 Previewing changes

Once, the pull request is submitted, GitHub Actions will automatically build the documentation. To view the built site:

1. Wait for the CI / Web and docs check to complete.
2. Click Details next to it.
3. From the Artifacts drop-down, click docs or website to download the site as a ZIP file.
4.4 Contributing to the code base

Table of Contents:

- Code standards
- Pre-commit
- Optional dependencies
  - C (cpplint)
  - Python (PEP8 / black)
  - Import formatting
  - Backwards compatibility
- Type hints
  - Style guidelines
  - pandas-specific types
  - Validating type hints
  - Testing type hints in code using pandas
- Testing with continuous integration
- Test-driven development/code writing
  - Writing tests
  - Transitioning to pytest
  - Using pytest
  - Using hypothesis
  - Testing warnings
- Running the test suite
- Running the performance test suite
- Documenting your code

4.4.1 Code standards

Writing good code is not just about what you write. It is also about how you write it. During Continuous Integration testing, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to pandas.

There is a tool in pandas to help contributors verify their changes before contributing them to the project:

```
./ci/code_checks.sh
```

The script validates the doctests, formatting in docstrings, static typing, and imported modules. It is possible to run the checks independently by using the parameters docstring, code, typing, and doctests (e.g. `./ci/code_checks.sh doctests`).
In addition, because a lot of people use our library, it is important that we do not make sudden changes to the code that could have the potential to break a lot of user code as a result, that is, we need it to be as backwards compatible as possible to avoid mass breakages.

In addition to ./ci/code_checks.sh, some extra checks are run by pre-commit - see here for how to run them.

Additional standards are outlined on the pandas code style guide.

### 4.4.2 Pre-commit

You can run many of these styling checks manually as we have described above. However, we encourage you to use pre-commit hooks instead to automatically run black, flake8, isort when you make a git commit. This can be done by installing pre-commit:

```
pip install pre-commit
```

and then running:

```
pre-commit install
```

from the root of the pandas repository. Now all of the styling checks will be run each time you commit changes without your needing to run each one manually. In addition, using pre-commit will also allow you to more easily remain up-to-date with our code checks as they change.

Note that if needed, you can skip these checks with git commit --no-verify.

If you don't want to use pre-commit as part of your workflow, you can still use it to run its checks with:

```
pre-commit run --files <files you have modified>
```

without needing to have done pre-commit install beforehand.

If you want to run checks on all recently committed files on upstream/main you can use:

```
pre-commit run --from-ref=upstream/main --to-ref=HEAD --all-files
```

without needing to have done pre-commit install beforehand.

**Note:** If you have conflicting installations of virtualenv, then you may get an error - see here.

Also, due to a bug in virtualenv, you may run into issues if you're using conda. To solve this, you can downgrade virtualenv to version 20.0.33.

### 4.4.3 Optional dependencies

Optional dependencies (e.g. matplotlib) should be imported with the private helper pandas.compat._optional.import_optional_dependency. This ensures a consistent error message when the dependency is not met.

All methods using an optional dependency should include a test asserting that an ImportError is raised when the optional dependency is not found. This test should be skipped if the library is present.

All optional dependencies should be documented in Optional dependencies and the minimum required version should be set in the pandas.compat._optional.VERSIONS dict.
C (cpplint)

pandas uses the Google standard. Google provides an open source style checker called cpplint, but we use a fork of it that can be found here. Here are some of the more common cpplint issues:

- we restrict line-length to 80 characters to promote readability
- every header file must include a header guard to avoid name collisions if re-included

Continuous Integration will run the cpplint tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-build/
         --include_subdir modified-c-file
```

You can also run this command on an entire directory if necessary:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-build/
         --include_subdir --recursive modified-c-directory
```

To make your commits compliant with this standard, you can install the ClangFormat tool, which can be downloaded here. To configure, in your home directory, run the following command:

```
clang-format style=google -dump-config > .clang-format
```

Then modify the file to ensure that any indentation width parameters are at least four. Once configured, you can run the tool as follows:

```
clang-format modified-c-file
```

This will output what your file will look like if the changes are made, and to apply them, run the following command:

```
clang-format -i modified-c-file
```

To run the tool on an entire directory, you can run the following analogous commands:

```
clang-format modified-c-directory/*.c modified-c-directory/*.h
clang-format -i modified-c-directory/*.c modified-c-directory/*.h
```

Do note that this tool is best-effort, meaning that it will try to correct as many errors as possible, but it may not correct all of them. Thus, it is recommended that you run cpplint to double check and make any other style fixes manually.

Python (PEP8 / black)

pandas follows the PEP8 standard and uses Black and Flake8 to ensure a consistent code format throughout the project. We encourage you to use pre-commit.

Continuous Integration will run those tools and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
black pandas
```

```
git diff upstream/main -u -- "*.py" | flake8 --diff
```

to auto-format your code. Additionally, many editors have plugins that will apply black as you edit files.

You should use a black version 21.5b2 as previous versions are not compatible with the pandas codebase.

4.4. Contributing to the code base
One caveat about `git diff upstream/main -u -- "*.py" | flake8 --diff`: this command will catch any stylistic errors in your changes specifically, but be beware it may not catch all of them. For example, if you delete the only usage of an imported function, it is stylistically incorrect to import an unused function. However, style-checking the diff will not catch this because the actual import is not part of the diff. Thus, for completeness, you should run this command, though it may take longer:

```
git diff upstream/main --name-only -- "*.py" | xargs -r flake8
```

Note that on macOS, the `-r` flag is not available, so you have to omit it and run this slightly modified command:

```
git diff upstream/main --name-only -- "*.py" | xargs flake8
```

Windows does not support the `xargs` command (unless installed for example via the MinGW toolchain), but one can imitate the behaviour as follows:

```
for /f %i in ('git diff upstream/main --name-only -- "*.py"') do flake8 %i
```

This will get all the files being changed by the PR (and ending with `.py`), and run `flake8` on them, one after the other. Note that these commands can be run analogously with `black`.

**Import formatting**

pandas uses `isort` to standardise import formatting across the codebase.

A guide to import layout as per pep8 can be found [here](#).

A summary of our current import sections (in order):

- Future
- Python Standard Library
- Third Party
- pandas._libs, pandas_compat, pandas_util.*, pandas.errors (largely not dependent on pandas.core)
- pandas.core.dtypes (largely not dependent on the rest of pandas.core)
- Rest of pandas.core.*
- Non-core pandas.io, pandas.plotting, pandas.tseries
- Local application/library specific imports

Imports are alphabetically sorted within these sections.

As part of Continuous Integration checks we run:

```
isort --check-only pandas
```

to check that imports are correctly formatted as per the `setup.cfg`.

If you see output like the below in Continuous Integration checks:

```
Check import format using isort
ERROR: /home/travis/build/pandas-dev/pandas/pandas/io/pytables.py Imports are incorrectly sorted
```

The `command "ci/code_checks.sh" exited with 1`
You should run:

```bash
isort pandas/io/pytables.py
```

to automatically format imports correctly. This will modify your local copy of the files.
Alternatively, you can run a command similar to what was suggested for black and flake8 right above:

```bash
git diff upstream/main --name-only -- "*.py" | xargs -r isort
```

Where similar caveats apply if you are on macOS or Windows.
You can then verify the changes look ok, then git commit and push.

**Backwards compatibility**

Please try to maintain backward compatibility. pandas has lots of users with lots of existing code, so don’t break it if at all possible. If you think breakage is required, clearly state why as part of the pull request. Also, be careful when changing method signatures and add deprecation warnings where needed. Also, add the deprecated sphinx directive to the deprecated functions or methods.

If a function with the same arguments as the one being deprecated exist, you can use the `pandas.util._decorators.deprecate`:

```python
from pandas.util._decorators import deprecate
deprecate('old_func', 'new_func', '1.1.0')
```

Otherwise, you need to do it manually:

```python
import warnings
def old_func():
    """Summary of the function.
    .. deprecated:: 1.1.0
       Use new_func instead.
    ""
    warnings.warn('Use new_func instead.', FutureWarning, stacklevel=2)
    new_func()

def new_func():
    pass
```

You’ll also need to

1. Write a new test that asserts a warning is issued when calling with the deprecated argument
2. Update all of pandas existing tests and code to use the new argument

See Testing warnings for more.
4.4.4 Type hints

pandas strongly encourages the use of PEP 484 style type hints. New development should contain type hints and pull requests to annotate existing code are accepted as well!

Style guidelines

Type imports should follow the from typing import ... convention. Some types do not need to be imported since PEP 585 some builtin constructs, such as list and tuple, can directly be used for type annotations. So rather than

```python
import typing
primes: typing.List[int] = []
```

You should write

```python
primes: list[int] = []
```

Optional should be avoided in favor of the shorter | None, so instead of

```python
from typing import Union
maybe_primes: list[Union[int, None]] = []
```

or

```python
from typing import Optional
maybe_primes: list[Optional[int]] = []
```

You should write

```python
from __future__ import annotations  # noqa: F404
maybe_primes: list[int | None] = []
```

In some cases in the code base classes may define class variables that shadow builtins. This causes an issue as described in Mypy 1775. The defensive solution here is to create an unambiguous alias of the builtin and use that without your annotation. For example, if you come across a definition like

```python
class SomeClass1:
    str = None
```

The appropriate way to annotate this would be as follows

```python
str_type = str
```

```python
class SomeClass2:
    str: str_type = None
```

In some cases you may be tempted to use cast from the typing module when you know better than the analyzer. This occurs particularly when using custom inference functions. For example
from typing import cast
from pandas.core.dtypes.common import is_number

def cannot_infer_bad(obj: Union[str, int, float]):
    if is_number(obj):
        ...
    else:  # Reasonably only str objects would reach this but...
        obj = cast(str, obj)  # Mypy complains without this!
        return obj.upper()

The limitation here is that while a human can reasonably understand that `is_number` would catch the `int` and `float` types mypy cannot make that same inference just yet (see mypy #5206. While the above works, the use of `cast` is strongly discouraged. Where applicable a refactor of the code to appease static analysis is preferable

def cannot_infer_good(obj: Union[str, int, float]):
    if isinstance(obj, str):
        return obj.upper()
    else:
        ...

With custom types and inference this is not always possible so exceptions are made, but every effort should be exhausted to avoid `cast` before going down such paths.

**pandas-specific types**

Commonly used types specific to pandas will appear in pandas._typing and you should use these where applicable. This module is private for now but ultimately this should be exposed to third party libraries who want to implement type checking against pandas.

For example, quite a few functions in pandas accept a `dtype` argument. This can be expressed as a string like "object", a `numpy.dtype` like `np.int64` or even a pandas `ExtensionDtype` like `pd.CategoricalDtype`. Rather than burden the user with having to constantly annotate all of those options, this can simply be imported and reused from the pandas._typing module

```python
from pandas._typing import Dtype

def as_type(dtype: Dtype) -> ...:
    ...
```

This module will ultimately house types for repeatedly used concepts like “path-like”, “array-like”, “numeric”, etc... and can also hold aliases for commonly appearing parameters like `axis`. Development of this module is active so be sure to refer to the source for the most up to date list of available types.
Validating type hints

pandas uses mypy and pyright to statically analyze the code base and type hints. After making any change you can ensure your type hints are correct by running

```makefile
mypy
# let pre-commit setup and run pyright
pre-commit run --hook-stage manual --all-files pyright
# or if pyright is installed (requires node.js)
pyright
```

A recent version of numpy (>=1.21.0) is required for type validation.

Testing type hints in code using pandas

**Warning:**
- Pandas is not yet a py.typed library (PEP 561)! The primary purpose of locally declaring pandas as a py.typed library is to test and improve the pandas-built-in type annotations.

Until pandas becomes a py.typed library, it is possible to easily experiment with the type annotations shipped with pandas by creating an empty file named “py.typed” in the pandas installation folder:

```python
python -c "import pandas; import pathlib; (pathlib.Path(pandas.__path__[0]) / 'py.typed').touch()"
```

The existence of the py.typed file signals to type checkers that pandas is already a py.typed library. This makes type checkers aware of the type annotations shipped with pandas.

4.4.5 Testing with continuous integration

The pandas test suite will run automatically on GitHub Actions and Azure Pipelines continuous integration services, once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then the continuous integration services need to be hooked to your GitHub repository. Instructions are here for GitHub Actions and Azure Pipelines.

A pull-request will be considered for merging when you have an all ‘green’ build. If any tests are failing, then you will get a red ‘X’, where you can click through to see the individual failed tests. This is an example of a green build.
4.4.6 Test-driven development/code writing

pandas is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process “relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test.” So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to pandas. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

Like many packages, pandas uses `pytest` and the convenient extensions in `numpy.testing`.

**Note:** The earliest supported pytest version is 5.0.1.

**Writing tests**

All tests should go into the `tests` subdirectory of the specific package. This folder contains many current examples of tests, and we suggest looking to these for inspiration. If your test requires working with files or network connectivity, there is more information on the wiki Testing of the wiki.

The `pandas._testing` module has many special `assert` functions that make it easier to make statements about whether Series or DataFrame objects are equivalent. The easiest way to verify that your code is correct is to explicitly construct the result you expect, then compare the actual result to the expected correct result:

```python
def test_pivot(self):
    data = {
        'index': ['A', 'B', 'C', 'C', 'B', 'A'],
        'columns': ['One', 'One', 'One', 'Two', 'Two', 'Two'],
        'values': [1., 2., 3., 3., 2., 1.]
    }

    frame = DataFrame(data)
```

(continues on next page)
```python
pivoted = frame.pivot(index='index', columns='columns', values='values')
expected = DataFrame({'One': {'A': 1., 'B': 2., 'C': 3.},
                      'Two': {'A': 1., 'B': 2., 'C': 3.}})
assert_frame_equal(pivoted, expected)
```

Please remember to add the Github Issue Number as a comment to a new test. E.g. “# brief comment, see GH#28907”

**Transitioning to pytest**

pandas existing test structure is *mostly* class-based, meaning that you will typically find tests wrapped in a class.

```python
class TestReallyCoolFeature:
    pass
```

Going forward, we are moving to a more *functional* style using the pytest framework, which offers a richer testing framework that will facilitate testing and developing. Thus, instead of writing test classes, we will write test functions like this:

```python
def test_really_cool_feature():
    pass
```

**Using pytest**

Here is an example of a self-contained set of tests that illustrate multiple features that we like to use.

- functional style: tests are like `test_*` and only take arguments that are either fixtures or parameters
- pytest.mark can be used to set metadata on test functions, e.g. skip or xfail.
- using parametrize: allow testing of multiple cases
- to set a mark on a parameter, pytest.param(..., marks=...) syntax should be used
- fixture, code for object construction, on a per-test basis
- using bare assert for scalars and truth-testing
- tm.assert_series_equal (and its counter part tm.assert_frame_equal), for pandas object comparisons.
- the typical pattern of constructing an expected and comparing versus the result

We would name this file `test_cool_feature.py` and put in an appropriate place in the pandas/tests/ structure.

```python
import pytest
import numpy as np
import pandas as pd

@pytest.mark.parametrize('dtype', [int8', 'int16', 'int32', 'int64']
def test_dtypes(dtype):
    assert str(np.dtype(dtype)) == dtype
```

(continues on next page)
@pytest.mark.parametrize(
    'dtype', ['float32', pytest.param('int16', marks=pytest.mark.skip),
              pytest.param('int32', marks=pytest.mark.xfail(reason='to show how it works'))]
)
def test_mark(dtype):
    assert str(np.dtype(dtype)) == 'float32'

@ pytest.fixture
def series():
    return pd.Series([1, 2, 3])

@ pytest.fixture(params=['int8', 'int16', 'int32', 'int64'])
def dtype(request):
    return request.param

def test_series(series, dtype):
    result = series.astype(dtype)
    assert result.dtype == dtype
    expected = pd.Series([1, 2, 3], dtype=dtype)
    tm.assert_series_equal(result, expected)

A test run of this yields

```
(pandas) bash-3.2$ pytest test_cool_feature.py -v
===================================================================
platform darwin -- Python 3.6.2, pytest-3.6.0, py-1.4.31, pluggy-0.4.0
collected 11 items

tester.py::test_dtypes[int8] PASSED
tester.py::test_dtypes[int16] PASSED
tester.py::test_dtypes[int32] PASSED
tester.py::test_dtypes[int64] PASSED
tester.py::test_mark[float32] PASSED
tester.py::test_mark[int16] SKIPPED
tester.py::test_mark[int32] xfail
tester.py::test_series[int8] PASSED
tester.py::test_series[int16] PASSED
tester.py::test_series[int32] PASSED
tester.py::test_series[int64] PASSED
```

Tests that we have parametrized are now accessible via the test name, for example we could run these with `-k int8` to sub-select *only* those tests which match int8.

```
(pandas) bash-3.2$ pytest test_cool_feature.py -v -k int8
===================================================================
platform darwin -- Python 3.6.2, pytest-3.6.0, py-1.4.31, pluggy-0.4.0
```
Using hypothesis

Hypothesis is a library for property-based testing. Instead of explicitly parametrizing a test, you can describe all valid inputs and let Hypothesis try to find a failing input. Even better, no matter how many random examples it tries, Hypothesis always reports a single minimal counterexample to your assertions - often an example that you would never have thought to test.

See Getting Started with Hypothesis for more of an introduction, then refer to the Hypothesis documentation for details.

```python
import json
from hypothesis import given, strategies as st

any_json_value = st.deferred(lambda: st.one_of(
    st.none(), st.booleans(), st.floats(allow_nan=False), st.text(),
    st.lists(any_json_value), st.dictionaries(st.text(), any_json_value)
))

@given(value=any_json_value)
def test_json_roundtrip(value):
    result = json.loads(json.dumps(value))
    assert value == result
```

This test shows off several useful features of Hypothesis, as well as demonstrating a good use-case: checking properties that should hold over a large or complicated domain of inputs.

To keep the pandas test suite running quickly, parametrized tests are preferred if the inputs or logic are simple, with Hypothesis tests reserved for cases with complex logic or where there are too many combinations of options or subtle interactions to test (or think of!) all of them.

Testing warnings

By default, one of pandas CI workers will fail if any unhandled warnings are emitted.

If your change involves checking that a warning is actually emitted, use `tm.assert_produces_warning(ExpectedWarning)`.

```python
import pandas._testing as tm
df = pd.DataFrame()
with tm.assert_produces_warning(FutureWarning):
    df.some_operation()
```

We prefer this to the `pytest.warns` context manager because ours checks that the warning’s stacklevel is set correctly. The stacklevel is what ensure the user’s file name and line number is printed in the warning, rather than something internal to pandas. It represents the number of function calls from user code (e.g. `df.some_operation()`) to the function that actually emits the warning. Our linter will fail the build if you use `pytest.warns` in a test.
If you have a test that would emit a warning, but you aren’t actually testing the warning itself (say because it’s going to be removed in the future, or because we’re matching a 3rd-party library’s behavior), then use `pytest.mark.filterwarnings` to ignore the error.

```python
@pytest.mark.filterwarnings("ignore:msg:category")
def test_thing(self):
...
```

If the test generates a warning of class `category` whose message starts with `msg`, the warning will be ignored and the test will pass.

If you need finer-grained control, you can use Python’s usual `warnings` module to control whether a warning is ignored / raised at different places within a single test.

```python
with warnings.catch_warnings():
    warnings.simplefilter("ignore", FutureWarning)
    # Or use warnings.filterwarnings(...) 
```

Alternatively, consider breaking up the unit test.

### 4.4.7 Running the test suite

The tests can then be run directly inside your Git clone (without having to install pandas) by typing:

```bash
pytest pandas
```

The tests suite is exhaustive and takes around 20 minutes to run. Often it is worth running only a subset of tests first around your changes before running the entire suite.

The easiest way to do this is with:

```bash
pytest pandas/path/to/test.py -k regex_matching_test_name
```

Or with one of the following constructs:

```bash
pytest pandas/tests/[test-module].py
pytest pandas/tests/[test-module].py::[TestClass]
pytest pandas/tests/[test-module].py::[TestClass]::[test_method]
```

Using `pytest-xdist`, one can speed up local testing on multicore machines. To use this feature, you will need to install pytest-xdist via:

```bash
pip install pytest-xdist
```

Two scripts are provided to assist with this. These scripts distribute testing across 4 threads.

On Unix variants, one can type:

```bash
test_fast.sh
```

On Windows, one can type:

```bash
test_fast.bat
```

This can significantly reduce the time it takes to locally run tests before submitting a pull request.

For more, see the `pytest` documentation.

### 4.4. Contributing to the code base

---
Furthermore one can run

```
pd.test()
```

with an imported pandas to run tests similarly.

### 4.4.8 Running the performance test suite

Performance matters and it is worth considering whether your code has introduced performance regressions. pandas is in the process of migrating to asv benchmarks to enable easy monitoring of the performance of critical pandas operations. These benchmarks are all found in the pandas/asv_bench directory, and the test results can be found here.

To use all features of asv, you will need either conda or virtualenv. For more details please check the asv installation webpage.

To install asv:

```
pip install git+https://github.com/airspeed-velocity/asv
```

If you need to run a benchmark, change your directory to asv_bench/ and run:

```
asv continuous -f 1.1 upstream/main HEAD
```

You can replace HEAD with the name of the branch you are working on, and report benchmarks that changed by more than 10%. The command uses conda by default for creating the benchmark environments. If you want to use virtualenv instead, write:

```
asv continuous -f 1.1 -E virtualenv upstream/main HEAD
```

The `-E virtualenv` option should be added to all asv commands that run benchmarks. The default value is defined in asv.conf.json.

Running the full benchmark suite can be an all-day process, depending on your hardware and its resource utilization. However, usually it is sufficient to paste only a subset of the results into the pull request to show that the committed changes do not cause unexpected performance regressions. You can run specific benchmarks using the `-b` flag, which takes a regular expression. For example, this will only run benchmarks from a pandas/asv_bench/benchmarks/groupby.py file:

```
asv continuous -f 1.1 upstream/main HEAD -b ^groupby
```

If you want to only run a specific group of benchmarks from a file, you can do it using .as a separator. For example:

```
asv continuous -f 1.1 upstream/main HEAD -b groupby.GroupByMethods
```

will only run the GroupByMethods benchmark defined in groupby.py.

You can also run the benchmark suite using the version of pandas already installed in your current Python environment. This can be useful if you do not have virtualenv or conda, or are using the setup.py develop approach discussed above; for the in-place build you need to set PYTHONPATH, e.g. `PYTHONPATH="$PWD/../" asv [remaining arguments]`. You can run benchmarks using an existing Python environment by:

```
asv run -e -E existing
```

or, to use a specific Python interpreter:
asv run -e -E existing:python3.6

This will display stderr from the benchmarks, and use your local python that comes from your $PATH.

Information on how to write a benchmark and how to use asv can be found in the asv documentation.

4.4.9 Documenting your code

Changes should be reflected in the release notes located in doc/source/whatsnew/vx.y.z.rst. This file contains an ongoing change log for each release. Add an entry to this file to document your fix, enhancement or (unavoidable) breaking change. Make sure to include the GitHub issue number when adding your entry (using :issue:`1234` where 1234 is the issue/pull request number). Your entry should be written using full sentences and proper grammar.

When mentioning parts of the API, use a Sphinx :func:, :meth:, or :class: directive as appropriate. Not all public API functions and methods have a documentation page; ideally links would only be added if they resolve. You can usually find similar examples by checking the release notes for one of the previous versions.

If your code is a bugfix, add your entry to the relevant bugfix section. Avoid adding to the Other section; only in rare cases should entries go there. Being as concise as possible, the description of the bug should include how the user may encounter it and an indication of the bug itself, e.g. “produces incorrect results” or “incorrectly raises”. It may be necessary to also indicate the new behavior.

If your code is an enhancement, it is most likely necessary to add usage examples to the existing documentation. This can be done following the section regarding documentation. Further, to let users know when this feature was added, the versionadded directive is used. The sphinx syntax for that is:

```
.. versionadded:: 1.1.0
```

This will put the text New in version 1.1.0 wherever you put the sphinx directive. This should also be put in the docstring when adding a new function or method (example) or a new keyword argument (example).

4.5 pandas code style guide

Table of contents:

- Patterns
- Testing
  - Failing tests
  - Do not use pytest.xfail
  - Using pytest.mark.xfail
- Miscellaneous
  - Reading from a url

pandas follows the PEP8 standard and uses Black and Flake8 to ensure a consistent code format throughout the project. We encourage you to use pre-commit to automatically run black, flake8, isort, and related code checks when you make a git commit.
4.5.1 Patterns

We use a flake8 plugin, pandas-dev-flaker, to check our codebase for unwanted patterns. See its README for the up-to-date list of rules we enforce.

4.5.2 Testing

Failing tests

See https://docs.pytest.org/en/latest/how-to/skipping.html for background.

Do not use pytest.xfail

Do not use this method. It has the same behavior as pytest.skip, namely it immediately stops the test and does not check if the test will fail. If this is the behavior you desire, use pytest.skip instead.

Using pytest.mark.xfail

Use this method if a test is known to fail but the manner in which it fails is not meant to be captured. It is common to use this method for a test that exhibits buggy behavior or a non-implemented feature. If the failing test has flaky behavior, use the argument strict=False. This will make it so pytest does not fail if the test happens to pass.

Prefer the decorator @pytest.mark.xfail and the argument pytest.param over usage within a test so that the test is appropriately marked during the collection phase of pytest. For xfailing a test that involves multiple parameters, a fixture, or a combination of these, it is only possible to xfail during the testing phase. To do so, use the request fixture:

```python
import pytest

def test_xfail(request):
    mark = pytest.mark.xfail(raising=TypeError, reason="Indicate why here")
    request.node.add_marker(mark)
```

xfail is not to be used for tests involving failure due to invalid user arguments. For these tests, we need to verify the correct exception type and error message is being raised, using pytest.raises instead.

4.5.3 Miscellaneous

Reading from a url

Good:

```python
from pandas.io.common import urlopen

with urlopen("http://www.google.com") as url:
    raw_text = url.read()
```
4.6 pandas maintenance

This guide is for pandas’ maintainers. It may also be interesting to contributors looking to understand the pandas development process and what steps are necessary to become a maintainer.

The main contributing guide is available at Contributing to pandas.

4.6.1 Roles

pandas uses two levels of permissions: triage and core team members.

Triage members can label and close issues and pull requests.

Core team members can label and close issues and pull request, and can merge pull requests.

GitHub publishes the full list of permissions.

4.6.2 Tasks

pandas is largely a volunteer project, so these tasks shouldn’t be read as “expectations” of triage and maintainers. Rather, they’re general descriptions of what it means to be a maintainer.

- Triage newly filed issues (see Issue triage)
- Review newly opened pull requests
- Respond to updates on existing issues and pull requests
- Drive discussion and decisions on stalled issues and pull requests
- Provide experience / wisdom on API design questions to ensure consistency and maintainability
- Project organization (run / attend developer meetings, represent pandas)

https://matthewrocklin.com/blog/2019/05/18/maintainer may be interesting background reading.

4.6.3 Issue triage

Here’s a typical workflow for triaging a newly opened issue.

1. Thank the reporter for opening an issue

   The issue tracker is many people’s first interaction with the pandas project itself, beyond just using the library. As such, we want it to be a welcoming, pleasant experience.

2. Is the necessary information provided?

   Ideally reporters would fill out the issue template, but many don’t. If crucial information (like the version of pandas they used), is missing feel free to ask for that and label the issue with “Needs info”. The report should follow the guidelines in Bug reports and enhancement requests. You may want to link to that if they didn’t follow the template.

   Make sure that the title accurately reflects the issue. Edit it yourself if it’s not clear.

3. Is this a duplicate issue?

   We have many open issues. If a new issue is clearly a duplicate, label the new issue as “Duplicate” assign the milestone “No Action”, and close the issue with a link to the original issue. Make sure to still thank the reporter, and encourage them to chime in on the original issue, and perhaps try to fix it.
If the new issue provides relevant information, such as a better or slightly different example, add it to the original issue as a comment or an edit to the original post.

4. **Is the issue minimal and reproducible?**

For bug reports, we ask that the reporter provide a minimal reproducible example. See https://matthewrocklin.com/blog/work/2018/02/28/minimal-bug-reports for a good explanation. If the example is not reproducible, or if it's clearly not minimal, feel free to ask the reporter if they can provide and example or simplify the provided one. Do acknowledge that writing minimal reproducible examples is hard work. If the reporter is struggling, you can try to write one yourself and we’ll edit the original post to include it.

If a reproducible example can’t be provided, add the “Needs info” label.

If a reproducible example is provided, but you see a simplification, edit the original post with your simpler reproducible example.

5. **Is this a clearly defined feature request?**

Generally, pandas prefers to discuss and design new features in issues, before a pull request is made. Encourage the submitter to include a proposed API for the new feature. Having them write a full docstring is a good way to pin down specifics.

We’ll need a discussion from several pandas maintainers before deciding whether the proposal is in scope for pandas.

6. **Is this a usage question?**

We prefer that usage questions are asked on StackOverflow with the pandas tag. https://stackoverflow.com/questions/tagged/pandas

If it’s easy to answer, feel free to link to the relevant documentation section, let them know that in the future this kind of question should be on StackOverflow, and close the issue.

7. **What labels and milestones should I add?**

Apply the relevant labels. This is a bit of an art, and comes with experience. Look at similar issues to get a feel for how things are labeled.

If the issue is clearly defined and the fix seems relatively straightforward, label the issue as “Good first issue”. Typically, new issues will be assigned the “Contributions welcome” milestone, unless it’s know that this issue should be addressed in a specific release (say because it’s a large regression).

### 4.6.4 Closing issues

Be delicate here: many people interpret closing an issue as us saying that the conversation is over. It’s typically best to give the reporter some time to respond or self-close their issue if it’s determined that the behavior is not a bug, or the feature is out of scope. Sometimes reporters just go away though, and we’ll close the issue after the conversation has died.
4.6.5 Reviewing pull requests

Anybody can review a pull request: regular contributors, triagers, or core-team members. But only core-team members can merge pull requests when they’re ready.

Here are some things to check when reviewing a pull request.

- Tests should be in a sensible location: in the same file as closely related tests.
- New public APIs should be included somewhere in `doc/source/reference/`.
- New / changed API should use the `versionadded` or `versionchanged` directives in the docstring.
- User-facing changes should have a `whatsnew` in the appropriate file.
- Regression tests should reference the original GitHub issue number like `# GH-1234`.
- The pull request should be labeled and assigned the appropriate milestone (the next patch release for regression fixes and small bug fixes, the next minor milestone otherwise)
- Changes should comply with our `Version policy`.

4.6.6 Backporting

In the case you want to apply changes to a stable branch from a newer branch then you can comment:

```
@meeseeksdev backport version-branch
```

This will trigger a workflow which will backport a given change to a branch (e.g. `@meeseeksdev backport 1.2.x`)

4.6.7 Cleaning up old issues

Every open issue in pandas has a cost. Open issues make finding duplicates harder, and can make it harder to know what needs to be done in pandas. That said, closing issues isn’t a goal on its own. Our goal is to make pandas the best it can be, and that’s best done by ensuring that the quality of our open issues is high.

Occasionally, bugs are fixed but the issue isn’t linked to in the Pull Request. In these cases, comment that “This has been fixed, but could use a test.” and label the issue as “Good First Issue” and “Needs Test”.

If an older issue doesn’t follow our issue template, edit the original post to include a minimal example, the actual output, and the expected output. Uniformity in issue reports is valuable.

If an older issue lacks a reproducible example, label it as “Needs Info” and ask them to provide one (or write one yourself if possible). If one isn’t provide reasonably soon, close it according to the policies in `Closing issues`.

4.6.8 Cleaning up old pull requests

Occasionally, contributors are unable to finish off a pull request. If some time has passed (two weeks, say) since the last review requesting changes, gently ask if they’re still interested in working on this. If another two weeks or so passes with no response, thank them for their work and close the pull request. Comment on the original issue that “There’s a stalled PR at #1234 that may be helpful.”, and perhaps label the issue as “Good first issue” if the PR was relatively close to being accepted.

Additionally, core-team members can push to contributors branches. This can be helpful for pushing an important PR across the line, or for fixing a small merge conflict.
### 4.6.9 Becoming a pandas maintainer

The full process is outlined in our governance documents. In summary, we’re happy to give triage permissions to anyone who shows interest by being helpful on the issue tracker.

The current list of core-team members is at https://github.com/pandas-dev/pandas-governance/blob/master/people.md

### 4.6.10 Merging pull requests

Only core team members can merge pull requests. We have a few guidelines.

1. You should typically not self-merge your own pull requests. Exceptions include things like small changes to fix CI (e.g. pinning a package version).
2. You should not merge pull requests that have an active discussion, or pull requests that has any −1 votes from a core maintainer. pandas operates by consensus.
3. For larger changes, it’s good to have a +1 from at least two core team members.

In addition to the items listed in Closing issues, you should verify that the pull request is assigned the correct milestone.

Pull requests merged with a patch-release milestone will typically be backported by our bot. Verify that the bot noticed the merge (it will leave a comment within a minute typically). If a manual backport is needed please do that, and remove the “Needs backport” label once you’ve done it manually. If you forget to assign a milestone before tagging, you can request the bot to backport it with:

@Meeseeksdev backport <branch>

### 4.7 Internals

This section will provide a look into some of pandas internals. It’s primarily intended for developers of pandas itself.

#### 4.7.1 Indexing

In pandas there are a few objects implemented which can serve as valid containers for the axis labels:

- **Index**: the generic “ordered set” object, an ndarray of object dtype assuming nothing about its contents. The labels must be hashable (and likely immutable) and unique. Populates a dict of label to location in Cython to do O(1) lookups.
- **Int64Index**: a version of Index highly optimized for 64-bit integer data, such as time stamps
- **Float64Index**: a version of Index highly optimized for 64-bit float data
- **MultiIndex**: the standard hierarchical index object
- **DatetimeIndex**: An Index object with Timestamp boxed elements (impl are the int64 values)
- **TimedeltaIndex**: An Index object with Timedelta boxed elements (impl are the int64 values)
- **PeriodIndex**: An Index object with Period elements

There are functions that make the creation of a regular index easy:

- **date_range**: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Python date-time objects
• period_range: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Period objects, representing timespans

The motivation for having an Index class in the first place was to enable different implementations of indexing. This means that it’s possible for you, the user, to implement a custom Index subclass that may be better suited to a particular application than the ones provided in pandas.

From an internal implementation point of view, the relevant methods that an Index must define are one or more of the following (depending on how incompatible the new object internals are with the Index functions):

• get_loc: returns an “indexer” (an integer, or in some cases a slice object) for a label
• slice_locs: returns the “range” to slice between two labels
• get_indexer: Computes the indexing vector for reindexing / data alignment purposes. See the source / docstrings for more on this
• get_indexer_non_unique: Computes the indexing vector for reindexing / data alignment purposes when the index is non-unique. See the source / docstrings for more on this
• reindex: Does any pre-conversion of the input index then calls get_indexer
• union, intersection: computes the union or intersection of two Index objects
• insert: Inserts a new label into an Index, yielding a new object
• delete: Delete a label, yielding a new object
• drop: Deletes a set of labels
• take: Analogous to ndarray.take

MultiIndex

Internally, the MultiIndex consists of a few things: the levels, the integer codes (until version 0.24 named labels), and the level names:

```
in [1]: index = pd.MultiIndex.from_product(
    ...:     [range(3), ["one", "two"], names=['first', 'second']
    ...:     )
    ...:

In [2]: index
Out[2]:
MultiIndex([(0, 'one'),
            (0, 'two'),
            (1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['first', 'second'])

In [3]: index.levels
Out[3]: FrozenList([0, 1, 2], ['one', 'two'])

In [4]: index.codes
Out[4]: FrozenList([[0, 0, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```
You can probably guess that the codes determine which unique element is identified with that location at each layer of the index. It's important to note that sortedness is determined solely from the integer codes and does not check (or care) whether the levels themselves are sorted. Fortunately, the constructors from_tuples and from_arrays ensure that this is true, but if you compute the levels and codes yourself, please be careful.

Values

pandas extends NumPy’s type system with custom types, like Categorical or datetimes with a timezone, so we have multiple notions of “values”. For 1-D containers (Index classes and Series) we have the following convention:

- cls._values refers is the “best possible” array. This could be an ndarray or ExtensionArray.

So, for example, Series[category]._values is a Categorical.

4.7.2 Subclassing pandas data structures

This section has been moved to Subclassing pandas data structures.

4.8 Test organization

Ideally, there should be one, and only one, obvious place for a test to reside. Until we reach that ideal, these are some rules of thumb for where a test should be located.

1. Does your test depend only on code in pd._libs.tslibs? This test likely belongs in one of:
   - tests.tslibs

   **Note:** No file in tests.tslibs should import from any pandas modules outside of pd._libs.tslibs

   - tests.scalar
   - tests.tseries.offsets

2. Does your test depend only on code in pd._libs? This test likely belongs in one of:
   - tests.libs
   - tests.groupby.test_libgroupby

3. Is your test for an arithmetic or comparison method? This test likely belongs in one of:
   - tests.arithmetic

   **Note:** These are intended for tests that can be shared to test the behavior of DataFrame/Series/Index/ExtensionArray using the box_with_array fixture.

   - tests.frame.test_arithmetic
   - tests.series.test_arithmetic
4. Is your test for a reduction method (min, max, sum, prod, …)? This test likely belongs in one of:
   - tests.reductions

   **Note:** These are intended for tests that can be shared to test the behavior of DataFrame/Series/Index/ExtensionArray.

   - tests.frame.test_reductions
   - tests.series.test_reductions
   - tests.test_nanops

5. Is your test for an indexing method? This is the most difficult case for deciding where a test belongs, because there are many of these tests, and many of them test more than one method (e.g. both Series.__getitem__ and Series.loc.__getitem__)
   
   A) Is the test specifically testing an Index method (e.g. Index.get_loc, Index.get_indexer)? This test likely belongs in one of:
      - tests.indexes.test_indexing
      - tests.indexes.fooindex.test_indexing

      Within that files there should be a method-specific test class e.g. TestGetLoc.

      In most cases, neither Series nor DataFrame objects should be needed in these tests.

   B) Is the test for a Series or DataFrame indexing method other than __getitem__ or __setitem__, e.g. xs, where, take, mask, lookup, or insert? This test likely belongs in one of:
      - tests.frame.indexing.test_methodname
      - tests.series.indexing.test_methodname

   C) Is the test for any of loc, iloc, at, or iat? This test likely belongs in one of:
      - tests.indexing.test_loc
      - tests.indexing.test_iloc
      - tests.indexing.test_at
      - tests.indexing.test_iat

      Within the appropriate file, test classes correspond to either types of indexers (e.g. TestLocBooleanMask) or major use cases (e.g. TestLocSetitemWithExpansion).

      See the note in section D) about tests that test multiple indexing methods.

   D) Is the test for Series.__getitem__, Series.__setitem__, DataFrame.__getitem__, or DataFrame.__setitem__? This test likely belongs in one of:
      - tests.series.test_getitem
      - tests.series.test_setitem
      - tests.frame.test_getitem
      - tests.frame.test_setitem

      If many cases such a test may test multiple similar methods, e.g.
import pandas as pd
import pandas._testing as tm

def test_getitem_listlike_of_ints():
    ser = pd.Series(range(5))

    result = ser[[3, 4]]
    expected = pd.Series([2, 3])
    tm.assert_series_equal(result, expected)

    result = ser.loc[[3, 4]]
    tm.assert_series_equal(result, expected)

In cases like this, the test location should be based on the underlying method being tested. Or in the case of a test for a bugfix, the location of the actual bug. So in this example, we know that Series.__getitem__ calls Series.loc.__getitem__, so this is really a test for loc.__getitem__. So this test belongs in tests.indexing.test_loc.

6. Is your test for a DataFrame or Series method?
   A) Is the method a plotting method? This test likely belongs in one of:
      • tests.plotting
   B) Is the method an IO method? This test likely belongs in one of:
      • tests.io
   C) Otherwise This test likely belongs in one of:
      • tests.series.methods.test_mymethod
      • tests.frame.methods.test_mymethod

   Note: If a test can be shared between DataFrame/Series using the frame_or_series fixture, by convention it goes in the tests.frame file.

7. Is your test for an Index method, not depending on Series/DataFrame? This test likely belongs in one of:
   • tests.indexes

8) Is your test for one of the pandas-provided ExtensionArrays (Categorical, DatetimeArray, TimedeltaArray, PeriodArray, IntervalArray, PandasArray, FloatArray, BoolArray, StringArray)? This test likely belongs in one of:
   • tests.arrays

9) Is your test for all ExtensionArray subclasses (the “EA Interface”)? This test likely belongs in one of:
   • tests.extension
4.9 Debugging C extensions

Pandas uses select C extensions for high performance IO operations. In case you need to debug segfaults or general issues with those extensions, the following steps may be helpful.

First, be sure to compile the extensions with the appropriate flags to generate debug symbols and remove optimizations. This can be achieved as follows:

```bash
python setup.py build_ext --inplace -j4 --with-debugging-symbols
```

4.9.1 Using a debugger

Assuming you are on a Unix-like operating system, you can use either lldb or gdb to debug. The choice between either is largely dependent on your compilation toolchain - typically you would use lldb if using clang and gdb if using gcc. For macOS users, please note that gcc is on modern systems an alias for clang, so if using Xcode you usually opt for lldb. Regardless of which debugger you choose, please refer to your operating systems instructions on how to install.

After installing a debugger you can create a script that hits the extension module you are looking to debug. For demonstration purposes, let’s assume you have a script called `debug_testing.py` with the following contents:

```python
import pandas as pd

pd.DataFrame([[1, 2]]).to_json()
```

Place the `debug_testing.py` script in the project root and launch a Python process under your debugger. If using lldb:

```
lldb python
```

If using gdb:

```
gdb python
```

Before executing our script, let’s set a breakpoint in our JSON serializer in its entry function called `objToJSON`. The lldb syntax would look as follows:

```lldb
breakpoint set --name objToJSON
```

Similarly for gdb:

```gdb
break objToJSON
```

**Note:** You may get a warning that this breakpoint cannot be resolved in lldb. gdb may give a similar warning and prompt you to make the breakpoint on a future library load, which you should say yes to. This should only happen on the very first invocation as the module you wish to debug has not yet been loaded into memory.

Now go ahead and execute your script:

```
run <the_script>.py
```

Code execution will halt at the breakpoint defined or at the occurrence of any segfault. LLDB’s [GDB to LLDB command map](https://lldb.llvm.org/manual/) provides a listing of debugger command that you can execute using either debugger.

Another option to execute the entire test suite under lldb would be to run the following:
Or for gdb

gdb --args python -m pytest

Once the process launches, simply type run and the test suite will begin, stopping at any segmentation fault that may occur.

### 4.9.2 Checking memory leaks with valgrind

You can use Valgrind to check for and log memory leaks in extensions. For instance, to check for a memory leak in a test from the suite you can run:

```
PYTHONMALLOC=malloc valgrind --leak-check=yes --track-origins=yes --log-file=valgrind-log.txt python -m pytest <path_to_a_test>
```

Note that code execution under valgrind will take much longer than usual. While you can run valgrind against extensions compiled with any optimization level, it is suggested to have optimizations turned off from compiled extensions to reduce the amount of false positives. The `--with-debugging-symbols` flag passed during package setup will do this for you automatically.

**Note:** For best results, you should run use a Python installation configured with Valgrind support (`--with-valgrind`)

### 4.10 Extending pandas

While pandas provides a rich set of methods, containers, and data types, your needs may not be fully satisfied. pandas offers a few options for extending pandas.

#### 4.10.1 Registering custom accessors

Libraries can use the decorators `pandas.api.extensions.register_dataframe_accessor()`, `pandas.api.extensions.register_series_accessor()`, and `pandas.api.extensions.register_index_accessor()`, to add additional “namespaces” to pandas objects. All of these follow a similar convention: you decorate a class, providing the name of attribute to add. The class’s `__init__` method gets the object being decorated. For example:

```python
@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._validate(pandas_obj)
        self._obj = pandas_obj

    @staticmethod
    def _validate(obj):
        # verify there is a column latitude and a column longitude
        if "latitude" not in obj.columns or "longitude" not in obj.columns:
            raise AttributeError("Must have 'latitude' and 'longitude'.")
```

(continues on next page)
@property
def center(self):
    # return the geographic center point of this DataFrame
    lat = self._obj.latitude
    lon = self._obj.longitude
    return (float(lon.mean()), float(lat.mean()))

def plot(self):
    # plot this array's data on a map, e.g., using Cartopy
    pass

Now users can access your methods using the geo namespace:

```python
>>> ds = pd.DataFrame(
    ...     {
        "longitude": np.linspace(0, 10),
        "latitude": np.linspace(0, 20)
    ...     }

>>> ds.geo.center
(5.0, 10.0)

>>> ds.geo.plot()
# plots data on a map
```

This can be a convenient way to extend pandas objects without subclassing them. If you write a custom accessor, make a pull request adding it to our ecosystem page.

We highly recommend validating the data in your accessor’s __init__. In our GeoAccessor, we validate that the data contains the expected columns, raising an AttributeError when the validation fails. For a Series accessor, you should validate the dtype if the accessor applies only to certain dtypes.

### 4.10.2 Extension types

| Warning | The pandas.api.extensions.ExtensionDtype and pandas.api.extensions.ExtensionArray APIs are new and experimental. They may change between versions without warning. |

pandas defines an interface for implementing data types and arrays that extend NumPy’s type system. pandas itself uses the extension system for some types that aren’t built into NumPy (categorical, period, interval, datetime with timezone). Libraries can define a custom array and data type. When pandas encounters these objects, they will be handled properly (i.e. not converted to an ndarray of objects). Many methods like pandas.isna() will dispatch to the extension type’s implementation.

If you’re building a library that implements the interface, please publicize it on ecosystem.extensions.

The interface consists of two classes.

---

4.10. Extending pandas
**ExtensionDtype**

A `pandas.api.extensions.ExtensionDtype` is similar to a `numpy.dtype` object. It describes the data type. Implementors are responsible for a few unique items like the name.

One particularly important item is the type property. This should be the class that is the scalar type for your data. For example, if you were writing an extension array for IP Address data, this might be `ipaddress.IPv4Address`.

See the extension dtype source for interface definition.

`pandas.api.extensions.ExtensionDtype` can be registered to pandas to allow creation via a string dtype name. This allows one to instantiate `Series` and `.astype()` with a registered string name, for example 'category' is a registered string accessor for the `CategoricalDtype`.

See the extension dtype dtypes for more on how to register dtypes.

**ExtensionArray**

This class provides all the array-like functionality. ExtensionArrays are limited to 1 dimension. An ExtensionArray is linked to an ExtensionDtype via the `dtype` attribute.

pandas makes no restrictions on how an extension array is created via its `__new__` or `__init__`, and puts no restrictions on how you store your data. We do require that your array be convertible to a NumPy array, even if this is relatively expensive (as it is for Categorical).

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 addresses may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists.

See the extension array source for the interface definition. The docstrings and comments contain guidance for properly implementing the interface.

**ExtensionArray operator support**

By default, there are no operators defined for the class `ExtensionArray`. There are two approaches for providing operator support for your ExtensionArray:

1. Define each of the operators on your `ExtensionArray` subclass.
2. Use an operator implementation from pandas that depends on operators that are already defined on the underlying elements (scalars) of the ExtensionArray.

**Note:** Regardless of the approach, you may want to set `__array_priority__` if you want your implementation to be called when involved in binary operations with NumPy arrays.

For the first approach, you define selected operators, e.g., `__add__`, `__le__`, etc. that you want your `ExtensionArray` subclass to support.

The second approach assumes that the underlying elements (i.e., scalar type) of the `ExtensionArray` have the individual operators already defined. In other words, if your `ExtensionArray` named `MyExtensionArray` is implemented so that each element is an instance of the class `MyExtensionElement`, then if the operators are defined for `MyExtensionElement`, the second approach will automatically define the operators for `MyExtensionArray`.

A mixin class, `ExtensionScalarOpsMixin` supports this second approach. If developing an `ExtensionArray` subclass, for example `MyExtensionArray`, can simply include `ExtensionScalarOpsMixin` as a parent class of
MyExtensionArray, and then call the methods _add_arithmetic_ops() and/or _add_comparison_ops() to hook the operators into your MyExtensionArray class, as follows:

```python
from pandas.api.extensions import ExtensionArray, ExtensionScalarOpsMixin

class MyExtensionArray(ExtensionArray, ExtensionScalarOpsMixin):
    pass

MyExtensionArray._add_arithmetic_ops()
MyExtensionArray._add_comparison_ops()
```

**Note:** Since pandas automatically calls the underlying operator on each element one-by-one, this might not be as performant as implementing your own version of the associated operators directly on the ExtensionArray.

For arithmetic operations, this implementation will try to reconstruct a new ExtensionArray with the result of the element-wise operation. Whether or not that succeeds depends on whether the operation returns a result that's valid for the ExtensionArray. If an ExtensionArray cannot be reconstructed, an ndarray containing the scalars returned instead.

For ease of implementation and consistency with operations between pandas and NumPy ndarrays, we recommend not handling Series and Indexes in your binary ops. Instead, you should detect these cases and return `NotImplemented`. When pandas encounters an operation like `op(Series, ExtensionArray)`, pandas will

1. unbox the array from the Series (Series.array)
2. call `result = op(values, ExtensionArray)`
3. re-box the result in a Series

**NumPy universal functions**

*Series* implements `__array_ufunc__`. As part of the implementation, pandas unboxes the ExtensionArray from the Series, applies the ufunc, and re-boxes it if necessary.

If applicable, we highly recommend that you implement `__array_ufunc__` in your extension array to avoid coercion to an ndarray. See the NumPy documentation for an example.

As part of your implementation, we require that you defer to pandas when a pandas container (*Series, DataFrame, Index*) is detected in inputs. If any of those is present, you should return `NotImplemented`. pandas will take care of unboxing the array from the container and re-calling the ufunc with the unwrapped input.

**Testing extension arrays**

We provide a test suite for ensuring that your extension arrays satisfy the expected behavior. To use the test suite, you must provide several pytest fixtures and inherit from the base test class. The required fixtures are found in [https://github.com/pandas-dev/pandas/blob/main/pandas/tests/extension/conftest.py](https://github.com/pandas-dev/pandas/blob/main/pandas/tests/extension/conftest.py).

To use a test, subclass it:

```python
from pandas.tests.extension import base
```

(continues on next page)
class TestConstructors(base.BaseConstructorsTests):
    pass

See https://github.com/pandas-dev/pandas/blob/main/pandas/tests/extension/base/__init__.py for a list of all the tests available.

**Compatibility with Apache Arrow**

An ExtensionArray can support conversion to / from pyarrow arrays (and thus support for example serialization to the Parquet file format) by implementing two methods: ExtensionArray.__arrow_array__ and ExtensionDtype.__from_arrow__.

The ExtensionArray.__arrow_array__ ensures that pyarrow knows how to convert the specific extension array into a pyarrow.Array (also when included as a column in a pandas DataFrame):

```python
class MyExtensionArray(ExtensionArray):
    ...

    def __arrow_array__(self, type=None):
        # convert the underlying array values to a pyarrow Array
        import pyarrow

        return pyarrow.array(..., type=type)
```

The ExtensionDtype.__from_arrow__ method then controls the conversion back from pyarrow to a pandas ExtensionArray. This method receives a pyarrow Array or ChunkedArray as only argument and is expected to return the appropriate pandas ExtensionArray for this dtype and the passed values:

```python
class ExtensionDtype:
    ...

    def __from_arrow__(self, array: pyarrow.Array/ChunkedArray) -> ExtensionArray:
        ...
```

See more in the Arrow documentation.

Those methods have been implemented for the nullable integer and string extension dtypes included in pandas, and ensure roundtrip to pyarrow and the Parquet file format.

### 4.10.3 Subclassing pandas data structures

**Warning:** There are some easier alternatives before considering subclassing pandas data structures.

1. Extensible method chains with *pipe*
2. Use *composition*. See here.
3. Extending by *registering an accessor*
4. Extending by *extension type*

This section describes how to subclass pandas data structures to meet more specific needs. There are two points that need attention:
1. Override constructor properties.
2. Define original properties

**Note:** You can find a nice example in geopandas project.

### Override constructor properties

Each data structure has several *constructor properties* for returning a new data structure as the result of an operation. By overriding these properties, you can retain subclasses through pandas data manipulations.

There are 3 possible constructor properties to be defined on a subclass:

- **DataFrame/Series._constructor**: Used when a manipulation result has the same dimension as the original.
- **DataFrame._constructor_sliced**: Used when a DataFrame (sub-)class manipulation result should be a Series (sub-)class.
- **Series._constructor_expanddim**: Used when a Series (sub-)class manipulation result should be a DataFrame (sub-)class, e.g. Series.to_frame().

Below example shows how to define SubclassedSeries and SubclassedDataFrame overriding constructor properties.

```python
class SubclassedSeries(pd.Series):
    @property
def _constructor(self):
        return SubclassedSeries

    @property
def _constructor_expanddim(self):
        return SubclassedDataFrame

class SubclassedDataFrame(pd.DataFrame):
    @property
def _constructor(self):
        return SubclassedDataFrame

    @property
def _constructor_sliced(self):
        return SubclassedSeries

>>> s = SubclassedSeries([1, 2, 3])
>>> type(s)
<class '__main__.SubclassedSeries'>

>>> to_framed = s.to_frame()
>>> type(to_framed)
<class '__main__.SubclassedDataFrame'>

>>> df = SubclassedDataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9

(continues on next page)
Define original properties

To let original data structures have additional properties, you should let pandas know what properties are added. pandas maps unknown properties to data names overriding __getattribute__. Defining original properties can be done in one of 2 ways:

1. Define _internal_names and _internal_names_set for temporary properties which WILL NOT be passed to manipulation results.
2. Define _metadata for normal properties which will be passed to manipulation results.

Below is an example to define two original properties, “internal_cache” as a temporary property and “added_property” as a normal property

```python
class SubclassedDataFrame2(pd.DataFrame):
    # temporary properties
    _internal_names = pd.DataFrame._internal_names + ["internal_cache"]
    _internal_names_set = set(_internal_names)

    # normal properties
    _metadata = ["added_property"]

    @property
    def _constructor(self):
        return SubclassedDataFrame2
```
>>> df = SubclassedDataFrame2({"A": [1, 2, 3], "B": [4, 5, 6], "C": [7, 8, 9]})
>>> df
          A  B  C
0     1  4  7
1     2  5  8
2     3  6  9

>>> df.internal_cache = "cached"
>>> df.added_property = "property"

>>> df.internal_cache
cached
>>> df.added_property
property

# properties defined in _internal_names is reset after manipulation
>>> df["A", "B"].internal_cache
AttributeError: 'SubclassedDataFrame2' object has no attribute 'internal_cache'

# properties defined in _metadata are retained
>>> df["A", "B"].added_property
property

### 4.10.4 Plotting backends

Starting in 0.25 pandas can be extended with third-party plotting backends. The main idea is letting users select a plotting backend different than the provided one based on Matplotlib. For example:

```python
>>> pd.set_option("plotting.backend", "backend.module")
>>> pd.Series([1, 2, 3]).plot()
```

This would be more or less equivalent to:

```python
>>> import backend.module
>>> backend.module.plot(pd.Series([1, 2, 3]))
```

The backend module can then use other visualization tools (Bokeh, Altair,...) to generate the plots.

Libraries implementing the plotting backend should use **entry points** to make their backend discoverable to pandas. The key is "pandas_plotting_backends". For example, pandas registers the default "matplotlib" backend as follows.

```python
# in setup.py
setup(
    ...,
    entry_points={
        "pandas_plotting_backends": [
            "matplotlib = pandas:plotting._matplotlib",
            ],
    },
)
```

More information on how to implement a third-party plotting backend can be found at [https://github.com/pandas-dev/pandas/blob/main/pandas/plotting/__init__.py#L1](https://github.com/pandas-dev/pandas/blob/main/pandas/plotting/__init__.py#L1).
4.11 Developer

This section will focus on downstream applications of pandas.

4.11.1 Storing pandas DataFrame objects in Apache Parquet format

The Apache Parquet format provides key-value metadata at the file and column level, stored in the footer of the Parquet file:

```python
5: optional list<KeyValue> key_value_metadata
```

where `KeyValue` is

```python
struct KeyValue {
  1: required string key
  2: optional string value
}
```

So that a `pandas.DataFrame` can be faithfully reconstructed, we store a `pandas` metadata key in the `FileMetaData` with the value stored as:

```json
{"index_columns": [<descr0>, <descr1>, ...],
 'column_indexes': [<ci0>, <ci1>, ..., <ciN>],
 'columns': [<c0>, <c1>, ...],
 'pandas_version': $VERSION,
 'creator': {
   'library': $LIBRARY,
   'version': $LIBRARY_VERSION
 }}
```

The “descriptor” values `<descr>` in the 'index_columns' field are strings (referring to a column) or dictionaries with values as described below.

The `<c0>`/<`ci0>` and so forth are dictionaries containing the metadata for each column, *including the index columns*. This has JSON form:

```json
{"name": column_name,
 'field_name': parquet_column_name,
 'pandas_type': pandas_type,
 'numpy_type': numpy_type,
 'metadata': metadata}
```

See below for the detailed specification for these.
Index metadata descriptors

RangeIndex can be stored as metadata only, not requiring serialization. The descriptor format for these as is follows:

```python
index = pd.RangeIndex(0, 10, 2)
{
    "kind": "range",
    "name": index.name,
    "start": index.start,
    "stop": index.stop,
    "step": index.step,
}
```

Other index types must be serialized as data columns along with the other DataFrame columns. The metadata for these is a string indicating the name of the field in the data columns, for example `__index_level_0__`.

If an index has a non-None name attribute, and there is no other column with a name matching that value, then the `index.name` value can be used as the descriptor. Otherwise (for unnamed indexes and ones with names colliding with other column names) a disambiguating name with pattern matching `__index_level_\d+__` should be used. In cases of named indexes as data columns, `name` attribute is always stored in the column descriptors as above.

Column metadata

`pandas_type` is the logical type of the column, and is one of:

- Boolean: 'bool'
- Integers: 'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64'
- Floats: 'float16', 'float32', 'float64'
- Date and Time Types: 'datetime', 'datetimetz', 'timedelta'
- String: 'unicode', 'bytes'
- Categorical: 'categorical'
- Other Python objects: 'object'

The `numpy_type` is the physical storage type of the column, which is the result of `str(dtype)` for the underlying NumPy array that holds the data. So for `datetimetz` this is `datetime64[ns]` and for categorical, it may be any of the supported integer categorical types.

The `metadata` field is None except for:

- `datetimetz`: {'timezone': zone, 'unit': 'ns'}. e.g. {'timezone': 'America/New_York', 'unit': 'ns'}. The 'unit' is optional, and if omitted it is assumed to be nanoseconds.
- `categorical`: {'num_categories': K, 'ordered': is_ordered, 'type': $TYPE}
  - Here 'type' is optional, and can be a nested pandas type specification here (but not categorical)
- `unicode`: {'encoding': encoding}
  - The encoding is optional, and if not present is UTF-8
- `object`: {'encoding': encoding}. Objects can be serialized and stored in `BYTE_ARRAY` Parquet columns. The encoding can be one of:
  - 'pickle'
  - 'bson'
- 'json'

- `timedelta`: `{ 'unit': 'ns' }`. The 'unit' is optional, and if omitted it is assumed to be nanoseconds. This metadata is optional altogether.

For types other than these, the 'metadata' key can be omitted. Implementations can assume None if the key is not present.

As an example of fully-formed metadata:

```json

{
    'index_columns': ['__index_level_0__'],
    'column_indexes': [
        {
            'name': None,
            'field_name': '__index_level_0__',
            'pandas_type': 'int64',
            'numpy_type': 'int64',
            'metadata': None
        },
        {
            'name': 'c0',
            'field_name': 'c0',
            'pandas_type': 'int8',
            'numpy_type': 'int8',
            'metadata': None
        },
        {
            'name': 'c1',
            'field_name': 'c1',
            'pandas_type': 'bytes',
            'numpy_type': 'object',
            'metadata': None
        },
        {
            'name': 'c2',
            'field_name': 'c2',
            'pandas_type': 'categorical',
            'numpy_type': 'int16',
            'metadata': { 'num_categories': 1000, 'ordered': False }
        },
        {
            'name': 'c3',
            'field_name': 'c3',
            'pandas_type': 'datetimetz',
            'numpy_type': 'datetime64[ns]',
            'metadata': { 'timezone': 'America/Los_Angeles' }
        },
        {
            'name': 'c4',
            'field_name': 'c4',
            'pandas_type': 'object',
            'numpy_type': 'object',
            'metadata': { 'encoding': 'pickle' }
        },
        {
            'name': None,
            'field_name': '__index_level_0__',
            'pandas_type': 'int64',
            'numpy_type': 'int64',
            'metadata': None
        }
    ],
    'pandas_version': '1.4.0',
    'creator': {
        'library': 'pyarrow',
        'version': '0.13.0'
    }
}
```
4.12 Policies

4.12.1 Version policy

Changed in version 1.0.0.

pandas uses a loose variant of semantic versioning (SemVer) to govern deprecations, API compatibility, and version numbering.

A pandas release number is made up of **MAJOR.MINOR.PATCH**.

API breaking changes should only occur in **major** releases. These changes will be documented, with clear guidance on what is changing, why it’s changing, and how to migrate existing code to the new behavior.

Whenever possible, a deprecation path will be provided rather than an outright breaking change.

pandas will introduce deprecations in **minor** releases. These deprecations will preserve the existing behavior while emitting a warning that provide guidance on:

- How to achieve similar behavior if an alternative is available
- The pandas version in which the deprecation will be enforced.

We will not introduce new deprecations in patch releases.

Deprecations will only be enforced in **major** releases. For example, if a behavior is deprecated in pandas 1.2.0, it will continue to work, with a warning, for all releases in the 1.x series. The behavior will change and the deprecation removed in the next major release (2.0.0).

---

**Note:** pandas will sometimes make behavior changing bug fixes, as part of minor or patch releases. Whether or not a change is a bug fix or an API-breaking change is a judgement call. We’ll do our best, and we invite you to participate in development discussion on the issue tracker or mailing list.

These policies do not apply to features marked as **experimental** in the documentation. pandas may change the behavior of experimental features at any time.

4.12.2 Python support

pandas will only drop support for specific Python versions (e.g. 3.6.x, 3.7.x) in pandas **major** or **minor** releases.

4.13 Roadmap

This page provides an overview of the major themes in pandas’ development. Each of these items requires a relatively large amount of effort to implement. These may be achieved more quickly with dedicated funding or interest from contributors.

An item being on the roadmap does not mean that it will necessarily happen, even with unlimited funding. During the implementation period we may discover issues preventing the adoption of the feature.

Additionally, an item not being on the roadmap does not exclude it from inclusion in pandas. The roadmap is intended for larger, fundamental changes to the project that are likely to take months or years of developer time. Smaller-scoped items will continue to be tracked on our issue tracker.

See **Roadmap evolution** for proposing changes to this document.
4.13.1 Extensibility

Pandas extension types allow for extending NumPy types with custom data types and array storage. Pandas uses extension types internally, and provides an interface for 3rd-party libraries to define their own custom data types.

Many parts of Pandas still unintentionally convert data to a NumPy array. These problems are especially pronounced for nested data.

We'd like to improve the handling of extension arrays throughout the library, making their behavior more consistent with the handling of NumPy arrays. We'll do this by cleaning up Pandas' internals and adding new methods to the extension array interface.

4.13.2 String data type

Currently, Pandas stores text data in an object dtype NumPy array. The current implementation has two primary drawbacks: First, object dtype is not specific to strings: any Python object can be stored in an object dtype array, not just strings. Second: this is not efficient. The NumPy memory model isn't especially well-suited to variable width text data.

To solve the first issue, we propose a new extension type for string data. This will initially be opt-in, with users explicitly requesting dtype="string". The array backing this string dtype may initially be the current implementation: an object dtype NumPy array of Python strings.

To solve the second issue (performance), we'll explore alternative in-memory array libraries (for example, Apache Arrow). As part of the work, we may need to implement certain operations expected by Pandas users (for example the algorithm used in Series.str.upper). That work may be done outside of Pandas.

4.13.3 Consistent missing value handling

Currently, Pandas handles missing data differently for different data types. We use different types to indicate that a value is missing (np.nan for floating-point data, np.nan or None for object-dtype data – typically strings or booleans – with missing values, and pd.NaT for datetimelike data). Integer data cannot store missing data or are cast to float. In addition, Pandas 1.0 introduced a new missing value sentinel, pd.NA, which is being used for the experimental nullable integer, boolean, and string data types.

These different missing values have different behaviors in user-facing operations. Specifically, we introduced different semantics for the nullable data types for certain operations (e.g. propagating in comparison operations instead of comparing as False).

Long term, we want to introduce consistent missing data handling for all data types. This includes consistent behavior in all operations (indexing, arithmetic operations, comparisons, etc.). There has been discussion of eventually making the new semantics the default.

This has been discussed at GH28095 (and linked issues), and described in more detail in this design doc.

4.13.4 Apache Arrow interoperability

Apache Arrow is a cross-language development platform for in-memory data. The Arrow logical types are closely aligned with typical Pandas use cases.

We'd like to provide better-integrated support for Arrow memory and data types within Pandas. This will let us take advantage of its I/O capabilities and provide for better interoperability with other languages and libraries using Arrow.
4.13.5 Block manager rewrite

We'd like to replace pandas current internal data structures (a collection of 1 or 2-D arrays) with a simpler collection of 1-D arrays.

pandas internal data model is quite complex. A DataFrame is made up of one or more 2-dimensional “blocks”, with one or more blocks per dtype. This collection of 2-D arrays is managed by the BlockManager.

The primary benefit of the BlockManager is improved performance on certain operations (construction from a 2D array, binary operations, reductions across the columns), especially for wide DataFrames. However, the BlockManager substantially increases the complexity and maintenance burden of pandas.

By replacing the BlockManager we hope to achieve

• Substantially simpler code
• Easier extensibility with new logical types
• Better user control over memory use and layout
• Improved micro-performance
• Option to provide a C / Cython API to pandas’ internals

See these design documents for more.

4.13.6 Decoupling of indexing and internals

The code for getting and setting values in pandas’ data structures needs refactoring. In particular, we must clearly separate code that converts keys (e.g., the argument to `DataFrame.loc`) to positions from code that uses these positions to get or set values. This is related to the proposed BlockManager rewrite. Currently, the BlockManager sometimes uses label-based, rather than position-based, indexing. We propose that it should only work with positional indexing, and the translation of keys to positions should be entirely done at a higher level.

Indexing is a complicated API with many subtleties. This refactor will require care and attention. More details are discussed at wiki (Tentative)-rules-for-restructuring-indexing-code

4.13.7 Numba-accelerated operations

Numba is a JIT compiler for Python code. We’d like to provide ways for users to apply their own Numba-jitted functions where pandas accepts user-defined functions (for example, `Series.apply()`, `DataFrame.apply()`, `DataFrame.applymap()`, and in groupby and window contexts). This will improve the performance of user-defined-functions in these operations by staying within compiled code.

4.13.8 Performance monitoring

pandas uses `airspeed velocity` to monitor for performance regressions. ASV itself is a fabulous tool, but requires some additional work to be integrated into an open source project’s workflow.

The `asv-runner` organization, currently made up of pandas maintainers, provides tools built on top of ASV. We have a physical machine for running a number of project’s benchmarks, and tools managing the benchmark runs and reporting on results.

We’d like to fund improvements and maintenance of these tools to

• Be more stable. Currently, they’re maintained on the nights and weekends when a maintainer has free time.
• Tune the system for benchmarks to improve stability, following https://pyperf.readthedocs.io/en/latest/system.html
• Build a GitHub bot to request ASV runs before a PR is merged. Currently, the benchmarks are only run nightly.

4.13.9 Roadmap evolution

pandas continues to evolve. The direction is primarily determined by community interest. Everyone is welcome to review existing items on the roadmap and to propose a new item.

Each item on the roadmap should be a short summary of a larger design proposal. The proposal should include

1. Short summary of the changes, which would be appropriate for inclusion in the roadmap if accepted.
2. Motivation for the changes.
3. An explanation of why the change is in scope for pandas.
4. Detailed design: Preferably with example-usage (even if not implemented yet) and API documentation
5. API Change: Any API changes that may result from the proposal.

That proposal may then be submitted as a GitHub issue, where the pandas maintainers can review and comment on the design. The pandas mailing list should be notified of the proposal.

When there’s agreement that an implementation would be welcome, the roadmap should be updated to include the summary and a link to the discussion issue.

4.13.10 Completed items

This section records now completed items from the pandas roadmap.

Documentation improvements

We improved the pandas documentation

• The pandas community worked with others to build the pydata-sphinx-theme, which is now used for https://pandas.pydata.org/docs/ (GH15556).
• Getting started contains a number of resources intended for new pandas users coming from a variety of backgrounds (GH26831).

4.14 Developer meetings

We hold regular developer meetings on the second Wednesday of each month at 18:00 UTC. These meetings and their minutes are open to the public. All are welcome to join.
4.14.1 Minutes

The minutes of past meetings are available in this Google Document.

4.14.2 Calendar

This calendar shows all the developer meetings.
You can subscribe to this calendar with the following links:

- iCal
- Google calendar

Additionally, we’ll sometimes have one-off meetings on specific topics. These will be published on the same calendar.
This is the list of changes to pandas between each release. For full details, see the commit logs. For install and upgrade instructions, see Installation.

5.1 Version 1.4

5.1.1 What's new in 1.4.2 (April 2, 2022)

These are the changes in pandas 1.4.2. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `DataFrame.drop()` and `Series.drop()` when `Index` had extension dtype and duplicates (GH45860)
- Fixed regression in `read_csv()` killing python process when invalid file input was given for `engine="c"` (GH45957)
- Fixed memory performance regression in `Series.fillna()` when called on a `DataFrame` column with `inplace=True` (GH46149)
- Provided an alternative solution for passing custom Excel formats in `Styler.to_excel()`, which was a regression based on stricter CSS validation. Examples available in the documentation for `Styler.format()` (GH46152)
- Fixed regression in `DataFrame.replace()` when a replacement value was also a target for replacement (GH46306)
- Fixed regression in `DataFrame.replace()` when the replacement value was explicitly None when passed in a dictionary to `to_replace` (GH45601, GH45836)
- Fixed regression when setting values with `DataFrame.loc()` losing `MultiIndex` names if `DataFrame` was empty before (GH46317)
- Fixed regression when rendering boolean datatype columns with `Styler()` (GH46384)
- Fixed regression in `Groupby.rolling()` with a frequency window that would raise a `ValueError` even if the datetimes within each group were monotonic (GH46061)
Bug fixes

- Fix some cases for subclasses that define their `_constructor` properties as general callables (GH46018)
- Fixed “longtable” formatting in `styler.to_latex()` when `column_format` is given in extended format (GH46037)
- Fixed incorrect rendering in `styler.format()` with `hyperlinks="html"` when the url contains a colon or other special characters (GH46389)
- Improved error message in `Rolling` when `window` is a frequency and `NaT` is in the rolling axis (GH46087)

Contributors

A total of 15 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Daniel Schmidt
- JHM Darbyshire
- Jonas Haag
- Jordan Hicks +
- Joris Van den Bossche
- Kian Eliasi +
- Luke Manley
- Marco Edward Gorelli
- Matthew Roeschke
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Richard Shadrach
- Simon Hawkins
- jbrockmendel

5.1.2 What’s new in 1.4.1 (February 12, 2022)

These are the changes in pandas 1.4.1. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Regression in `Series.mask()` with `inplace=True` and `PeriodDtype` and an incompatible other coercing to a common dtype instead of raising (GH45546)
- Regression in `assert_frame_equal()` not respecting `check_flags=False` (GH45554)
- Regression in `DataFrame.loc()` raising `ValueError` when indexing (getting values) on a `MultiIndex` with one level (GH45779)
- Regression in `Series.fillna()` with `downcast=False` incorrectly downcasting object dtype (GH45603)
- Regression in `api.types.is_bool_dtype()` raising an `AttributeError` when evaluating a categorical `Series` (GH45615)
- Regression in `DataFrame.iat()` setting values leading to not propagating correctly in subsequent lookups (GH45684)
- Regression when setting values with `DataFrame.loc()` losing `Index` name if `DataFrame` was empty before (GH45621)
- Regression in `join()` with overlapping `IntervalIndex` raising an `InvalidIndexError` (GH45661)
- Regression when setting values with `Series.loc()` raising with all `False` indexer and `Series` on the right hand side (GH45778)
- Regression in `read_sql()` with a DBAPI2 connection that is not an instance of `sqlite3.Connection` incorrectly requiring SQLAlchemy be installed (GH45660)
- Regression in `DateOffset` when constructing with an integer argument with no keywords (e.g. `pd.DateOffset(n)`) would behave like `datetime.timedelta(days=0)` (GH45643, GH45890)

Bug fixes

- Fixed segfault in `DataFrame.to_json()` when dumping tz-aware datetimes in Python 3.10 (GH42130)
- Stopped emitting unnecessary `FutureWarning` in `DataFrame.sort_values()` with sparse columns (GH45618)
- Fixed window aggregations in `DataFrame.rolling()` and `Series.rolling()` to skip over unused elements (GH45647)
- Fixed builtin highlighters in `Styler` to be responsive to `NA` with nullable dtypes (GH45804)
- Bug in `apply()` with `axis=1` raising an erroneous `ValueError` (GH45912)

Other

- Reverted performance speedup of `DataFrame.corr()` for `method=pearson` to fix precision regression (GH45640, GH42761)
Contributors

A total of 19 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam Bowden +
- JHM Darbyshire
- Johannes Mueller
- John Zangwill
- Jonas Haag
- Matthew Roeschke
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Robbie Palmer +
- Simon Hawkins
- Thomas Li
- Tyler Reddy
- eshirvana +
- gesoos +
- hasan-yaman
- jbrockmendel
- partev
- rtpsw +

5.1.3 What’s new in 1.4.0 (January 22, 2022)

These are the changes in pandas 1.4.0. See Release notes for a full changelog including other versions of pandas.

Enhancements

Improved warning messages

Previously, warning messages may have pointed to lines within the pandas library. Running the script setting_with_copy_warning.py

```python
import pandas as pd

df = pd.DataFrame({'a': [1, 2, 3]})
df[2].loc[:, 'a'] = 5
```

with pandas 1.3 resulted in:
.../site-packages/pandas/core/indexing.py:1951: SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame.
This made it difficult to determine where the warning was being generated from. Now pandas will inspect the call stack, reporting the first line outside of the pandas library that gave rise to the warning. The output of the above script is now:

setting_with_copy_warning.py:4: SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame.

Index can hold arbitrary ExtensionArrays

Until now, passing a custom ExtensionArray to pd.Index would cast the array to object dtype. Now Index can directly hold arbitrary ExtensionArrays (GH43930).

Previous behavior:

```python
In [1]: arr = pd.array([1, 2, pd.NA])
In [2]: idx = pd.Index(arr)
```

In the old behavior, `idx` would be object-dtype:

```python
Previous behavior:
In [1]: idx
Out[1]: Index([1, 2, <NA>], dtype='object')
```

With the new behavior, we keep the original dtype:

```python
New behavior:
In [3]: idx
Out[3]: Index([1, 2, <NA>], dtype='Int64')
```

One exception to this is SparseArray, which will continue to cast to numpy dtype until pandas 2.0. At that point it will retain its dtype like other ExtensionArrays.

Styler

Styler has been further developed in 1.4.0. The following general enhancements have been made:

- Styling and formatting of indexes has been added, with Styler.apply_index(), Styler.applymap_index() and Styler.format_index(). These mirror the signature of the methods already used to style and format data values, and work with both HTML, LaTeX and Excel format (GH41893, GH43101, GH41993, GH41995)
- The new method Styler.hide() deprecates Styler.hide_index() and Styler.hide_columns() (GH43758)
- The keyword arguments level and names have been added to Styler.hide() (and implicitly to the deprecated methods Styler.hide_index() and Styler.hide_columns()) for additional control of visibility of MultiIndexes and of Index names (GH25475, GH43404, GH43346)
- The Styler.export() and Styler.use() have been updated to address all of the added functionality from v1.2.0 and v1.3.0 (GH40675)
Global options under the category `pd.options.styler` have been extended to configure default `Styler` properties which address formatting, encoding, and HTML and LaTeX rendering. Note that formerly `Styler` relied on `display.html.use_mathjax`, which has now been replaced by `styler.html.mathjax` (GH41395).

Validation of certain keyword arguments, e.g. `caption` (GH43368).

Various bug fixes as recorded below.

Additionally there are specific enhancements to the HTML specific rendering:

- `Styler.bar()` introduces additional arguments to control alignment and display (GH26070, GH36419), and it also validates the input arguments `width` and `height` (GH42511).
- `Styler.to_html()` introduces keyword arguments `sparse_index`, `sparse_columns`, `bold_headers`, `caption`, `max_rows` and `max_columns` (GH41946, GH43149, GH42972).
- `Styler.to_html()` omits CSSStyle rules for hidden table elements as a performance enhancement (GH43619).
- Custom CSS classes can now be directly specified without string replacement (GH43686).
- Ability to render hyperlinks automatically via a new `hyperlinks` formatting keyword argument (GH45058).

There are also some LaTeX specific enhancements:

- `Styler.to_latex()` introduces keyword argument `environment`, which also allows a specific “longtable” entry through a separate jinja2 template (GH41866).
- Naive sparsification is now possible for LaTeX without the necessity of including the multirow package (GH43369).
- `cline` support has been added for `MultiIndex` row sparsification through a keyword argument (GH45138).

Multi-threaded CSV reading with a new CSV Engine based on pyarrow

`pandas.read_csv()` now accepts `engine="pyarrow"` (requires at least pyarrow 1.0.1) as an argument, allowing for faster csv parsing on multicore machines with pyarrow installed. See the I/O docs for more info. (GH23697, GH43706)

Rank function for rolling and expanding windows

Added rank function to Rolling and Expanding. The new function supports the method, ascending, and pct flags of `DataFrame.rank()`. The method argument supports min, max, and average ranking methods. Example:

```
In [4]: s = pd.Series([1, 4, 2, 3, 5, 3])

In [5]: s.rolling(3).rank()
Out[5]:
      0   NaN
      1   NaN
      2   2.0
      3   2.0
      4   3.0
      5   1.5
dtype: float64

In [6]: s.rolling(3).rank(method="max")
Out[6]:
      0   NaN
```

Groupby positional indexing

It is now possible to specify positional ranges relative to the ends of each group. Negative arguments for `GroupBy.head()` and `GroupBy.tail()` now work correctly and result in ranges relative to the end and start of each group, respectively. Previously, negative arguments returned empty frames.

```
In [7]: df = pd.DataFrame([["g", "g0"], ["g", "g1"], ["g", "g2"], ["g", "g3"],
                        ["h", "h0"], ["h", "h1"]], columns=["A", "B"])
...

In [8]: df.groupby("A").head(-1)
Out[8]:
   A  B
0  g  g0
1  g  g1
2  g  g2
4  h  h0
```

`GroupBy.nth()` now accepts a slice or list of integers and slices.

```
In [9]: df.groupby("A").nth(slice(1, -1))
Out[9]:
   B
A  
g  g1
 g  g2

In [10]: df.groupby("A").nth([slice(None, 1), slice(-1, None)])
Out[10]:
   B
A  
g  g0
 g  g3
 h  h0
 h  h1
```

`GroupBy.nth()` now accepts index notation.

```
In [11]: df.groupby("A").nth[1, -1]
Out[11]:
   B
A  
g  g1
 g  g3
```
DataFrame.from_dict and DataFrame.to_dict have new 'tight' option

A new 'tight' dictionary format that preserves MultiIndex entries and names is now available with the DataFrame.from_dict() and DataFrame.to_dict() methods and can be used with the standard json library to produce a tight representation of DataFrame objects (GH4889).

```python
In [14]: df = pd.DataFrame.from_records(
     ....:     [[1, 3], [2, 4]],
     ....:     index=pd.MultiIndex.from_tuples([('a', 'b'), ('a', 'c')],
     ....:     names=['n1', 'n2']),
     ....:     columns=pd.MultiIndex.from_tuples([('x', 1), ('y', 2)],
     ....:     names=['z1', 'z2']),
     ....:   )

In [15]: df
Out[15]:
   z1  x  y
   z2
   n1 n2
   a  b  1  3
   c  2  4

In [16]: df.to_dict(orient='tight')
Out[16]:
{'index': [('a', 'b'), ('a', 'c')],
 'columns': [('x', 1), ('y', 2)],
 'data': [[1, 3], [2, 4]],
 'index_names': ['n1', 'n2'],
 'column_names': ['z1', 'z2']}
```
Other enhancements

- `concat()` will preserve the `attrs` when it is the same for all objects and discard the `attrs` when they are different (GH41828)
- `DataFrameGroupBy` operations with `as_index=False` now correctly retain `ExtensionDtype` dtypes for columns being grouped on (GH41373)
- Add support for assigning values to by argument in `DataFrame.plot.hist()` and `DataFrame.plot.box()` (GH15079)
- `Series.sample()`, `DataFrame.sample()`, and `GroupBy.sample()` now accept a `np.random.Generator` as input to `random_state`. A generator will be more performant, especially with `replace=False` (GH38100)
- `Series.ewm()` and `DataFrame.ewm()` now support a method argument with a 'table' option that performs the windowing operation over an entire `DataFrame`. See Window Overview for performance and functional benefits (GH42273)
- `GroupBy.cummin()` and `GroupBy.cummax()` now support the argument `skipna` (GH34047)
- `read_table()` now supports the argument `storage_options` (GH39167)
- `DataFrame.to_stata()` and `StataWriter()` now accept the keyword only argument `value_labels` to save labels for non-categorical columns (GH38454)
- Methods that relied on hashmap based algos such as `DataFrameGroupBy.value_counts()`, `DataFrameGroupBy.count()` and `factorize()` ignored imaginary component for complex numbers (GH17927)
- Add `Series.str.removeprefix()` and `Series.str.removesuffix()` introduced in Python 3.9 to remove pre-/suffixes from string-type `Series` (GH36944)
- Attempting to write into a file in missing parent directory with `DataFrame.to_csv()`, `DataFrame.to_html()`, `DataFrame.to_excel()`, `DataFrame.to_feather()`, `DataFrame.to_parquet()`, `DataFrame.to_stata()`, `DataFrame.to_json()`, `DataFrame.to_pickle()`, and `DataFrame.to_xml()` now explicitly mentions missing parent directory, the same is true for `Series` counterparts (GH24306)
- Indexing with `.loc` and `.iloc` now supports Ellipsis (GH37750)
- `IntegerArray.all()`, `IntegerArray.any()`, `FloatingArray.any()`, and `FloatingArray.all()` use Kleene logic (GH41967)
- Added support for nullable boolean and integer types in `DataFrame.to_stata()`, `StataWriter`, `StataWriter117`, and `StataWriterUTF8` (GH40855)
- `DataFrame.__pos__()` and `DataFrame.__neg__()` now retain `ExtensionDtype` dtypes (GH43883)
- The error raised when an optional dependency can't be imported now includes the original exception, for easier investigation (GH43882)
- Added `ExponentialMovingWindow.sum()` (GH13297)
- `Series.str.split()` now supports a `regex` argument that explicitly specifies whether the pattern is a regular expression. Default is `None` (GH43564, GH32835, GH25549)
- `DataFrame.dropna()` now accepts a single label as `subset` along with array-like (GH41021)
- Added `DataFrameGroupBy.value_counts()` (GH43564)
- `read_csv()` now accepts a callable function in `on_bad_lines` when `engine="python"` for custom handling of bad lines (GH5686)
- `ExcelWriter` argument `if_sheet_exists="overlay"` option added (GH40231)
**read_excel()** now accepts a `decimal` argument that allow the user to specify the decimal point when parsing string columns to numeric (GH14403)

- `GroupBy.mean()`, `GroupBy.std()`, `GroupBy.var()`, and `GroupBy.sum()` now support Numba execution with the engine keyword (GH43731, GH44862, GH44939)

- `Timestamp.isoformat()` now handles the `timespec` argument from the base `datetime` class (GH26131)

- `NaT.to_numpy()` `dtype` argument is now respected, so `np.timedelta64` can be returned (GH44460)

- New option `display.max_dir_items` customizes the number of columns added to `Dataframe.__dir__()` and suggested for tab completion (GH37996)

- Added “Juneteenth National Independence Day” to `USFederalHolidayCalendar` (GH44574)

- `Rolling.var()`, `Expanding.var()`, `Rolling.std()`, and `Expanding.std()` now support Numba execution with the engine keyword (GH44461)

- `Series.info()` has been added, for compatibility with `DataFrame.info()` (GH5167)

- Implemented `IntervalArray.min()` and `IntervalArray.max()`, as a result of which `min` and `max` now work for `IntervalIndex`, `Series` and `DataFrame` with `IntervalDtype` (GH44746)

- `UInt64Index.map()` now retains `dtype` where possible (GH44609)

- `read_json()` can now parse unsigned long long integers (GH26068)

- `DataFrame.take()` now raises a `TypeError` when passed a scalar for the indexer (GH42875)

- `is_list_like()` now identifies duck-arrays as list-like unless `.ndim == 0` (GH35131)

- `ExtensionDtype` and `ExtensionArray` are now (de)serialized when exporting a `DataFrame` with `DataFrame.to_json()` using `orient='table'` (GH20612, GH44705)

- Add support for Zstandard compression to `DataFrame.to_pickle()`/`read_pickle()` and friends (GH43925)

- `DataFrame.to_sql()` now returns an `int` of the number of written rows (GH23998)

### Notable bug fixes

These are bug fixes that might have notable behavior changes.

### Inconsistent date string parsing

The `dayfirst` option of `to_datetime()` isn’t strict, and this can lead to surprising behavior:

```
In [17]: pd.to_datetime("31-12-2021", dayfirst=False)
Out[17]: DatetimeIndex(['2021-12-31'], dtype='datetime64[ns]', freq=None)
```

Now, a warning will be raised if a date string cannot be parsed accordance to the given `dayfirst` value when the value is a delimited date string (e.g. `31-12-2012`).
Ignoring dtypes in concat with empty or all-NA columns

When using `concat()` to concatenate two or more `DataFrame` objects, if one of the DataFrames was empty or had all-NA values, its dtype was *sometimes* ignored when finding the concatenated dtype. These are now consistently *not* ignored (GH43507).

```python
In [18]: df1 = pd.DataFrame({"bar": [pd.Timestamp("2013-01-01")], index=range(1))
In [19]: df2 = pd.DataFrame({"bar": np.nan}, index=range(1, 2))
In [20]: res = pd.concat([df1, df2])
```

Previously, the float-dtype in `df2` would be ignored so the result dtype would be `datetime64[ns]`. As a result, the `np.nan` would be cast to `NaT`.

*Previous behavior:*

```python
In [4]: res
Out[4]:
bar
0 2013-01-01
1 NaT
```

Now the float-dtype is respected. Since the common dtype for these DataFrames is object, the `np.nan` is retained.

*New behavior:*

```python
In [21]: res
Out[21]:
bar
0 2013-01-01 00:00:00
1 NaN
```

Null-values are no longer coerced to NaN-value in value_counts and mode

`Series.value_counts()` and `Series.mode()` no longer coerce `None`, `NaT` and other null-values to a `NaN`-value for `np.object`-dtype. This behavior is now consistent with `unique`, `isin` and others (GH42688).

```python
In [22]: s = pd.Series([True, None, pd.NaT, None, pd.NaT, None])
In [23]: res = s.value_counts(dropna=False)
```

Previously, all null-values were replaced by a `NaN`-value.

*Previous behavior:*

```python
In [3]: res
Out[3]:
NaN 5
True 1
dtype: int64
```

Now null-values are no longer mangled.

*New behavior:*
## mangle_dupe_cols in read_csv no longer renames unique columns conflicting with target names

`read_csv()` no longer renames unique column labels which conflict with the target names of duplicated columns. Already existing columns are skipped, i.e. the next available index is used for the target column name (GH14704).

```python
In [25]: import io
In [26]: data = "a,a,a.1\n   \n   1,2,3"
In [27]: res = pd.read_csv(io.StringIO(data))
```

Previously, the second column was called `a.1`, while the third column was also renamed to `a.1.1`.

**Previous behavior:**

```python
In [3]: res
Out[3]:
   a  a.1  a.1.1
0  1   2    3
```

Now the renaming checks if `a.1` already exists when changing the name of the second column and jumps this index. The second column is instead renamed to `a.2`.

**New behavior:**

```python
In [28]: res
Out[28]:
   a  a.2  a.1
0  1   2    3
```

## unstack and pivot_table no longer raises ValueError for result that would exceed int32 limit

Previously `DataFrame.pivot_table()` and `DataFrame.unstack()` would raise a `ValueError` if the operation could produce a result with more than `2**31 - 1` elements. This operation now raises a `errors.PerformanceWarning` instead (GH26314).

**Previous behavior:**

```python
In [3]: df = DataFrame({"ind1": np.arange(2 ** 16), "ind2": np.arange(2 ** 16), "count": ... 0})
In [4]: df.pivot_table(index="ind1", columns="ind2", values="count", aggfunc="count")
ValueError: Unstacked DataFrame is too big, causing int32 overflow
```

**New behavior:**

---

2866 Chapter 5. Release notes
In [4]: df.pivot_table(index="ind1", columns="ind2", values="count", aggfunc="count")
PerformanceWarning: The following operation may generate 4294967296 cells in the resulting pandas object.

**groupby.apply consistent transform detection**

*GroupBy.apply()* is designed to be flexible, allowing users to perform aggregations, transformations, filters, and use it with user-defined functions that might not fall into any of these categories. As part of this, apply will attempt to detect when an operation is a transform, and in such a case, the result will have the same index as the input. In order to determine if the operation is a transform, pandas compares the input’s index to the result’s and determines if it has been mutated. Previously in pandas 1.3, different code paths used different definitions of “mutated”: some would use Python’s *is* whereas others would test only up to equality.

This inconsistency has been removed, pandas now tests up to equality.

```
In [29]: def func(x):
    ....:     return x.copy()
    ....:
In [30]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [5, 6]})
In [31]: df
Out[31]:
   a  b  c
0  1  3  5
1  2  4  6
```

**Previous behavior:**

```
In [3]: df.groupby(['a']).apply(func)
Out[3]:
   a  b  c
a 
1  0  1  3  5
2  1  2  4  6
```

```
In [4]: df.set_index(['a', 'b']).groupby(['a']).apply(func)
Out[4]:
   c
a  b
1  3  5
2  4  6
```

In the examples above, the first uses a code path where pandas uses *is* and determines that *func* is not a transform whereas the second tests up to equality and determines that *func* is a transform. In the first case, the result’s index is not the same as the input’s.

**New behavior:**

```
In [32]: df.groupby(['a']).apply(func)
Out[32]:
   a  b  c
0  1  3  5
```

(continues on next page)
In [33]: df.set_index(['a', 'b']).groupby('a').apply(func)
Out[33]:
   c
a b
1 3 5
2 4 6

Now in both cases it is determined that func is a transform. In each case, the result has the same index as the input.

**Backwards incompatible API changes**

**Increased minimum version for Python**

pandas 1.4.0 supports Python 3.8 and higher.

**Increased minimum versions for dependencies**

Some minimum supported versions of dependencies were updated. If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.18.5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pytz</td>
<td>2020.1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.8.1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.3.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.7.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mypy (dev)</td>
<td>0.930</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.
## Other API changes

- **Index.get_indexer_for()** no longer accepts keyword arguments (other than target); in the past these would be silently ignored if the index was not unique (GH42310)
- Change in the position of the min_rows argument in `DataFrame.to_string()` due to change in the docstring (GH44304)
- Reduction operations for DataFrame or Series now raising a ValueError when None is passed for skipna (GH44178)
- `read_csv()` and `read_html()` no longer raising an error when one of the header rows consists only of Unnamed: columns (GH13054)
- Changed the name attribute of several holidays in USFederalHolidayCalendar to match official federal holiday names specifically:
  - “New Year’s Day” gains the possessive apostrophe
  - “Presidents Day” becomes “Washington’s Birthday”
  - “Martin Luther King Jr. Day” is now “Birthday of Martin Luther King, Jr.”
  - “July 4th” is now “Independence Day”
  - “Thanksgiving” is now “Thanksgiving Day”
  - “Christmas” is now “Christmas Day”
  - Added “Juneteenth National Independence Day”
Deprecations

**Deprecated Int64Index, UInt64Index & Float64Index**

*Int64Index, UInt64Index* and *Float64Index* have been deprecated in favor of the base *Index* class and will be removed in Pandas 2.0 (GH43028).

For constructing a numeric index, you can use the base *Index* class instead specifying the data type (which will also work on older pandas releases):

```
# replace
pd.Int64Index([1, 2, 3])
# with
pd.Index([1, 2, 3], dtype="int64")
```

For checking the data type of an index object, you can replace *isinstance* checks with checking the *dtype*:

```
# replace
isinstance(idx, pd.Int64Index)
# with
idx.dtype == "int64"
```

Currently, in order to maintain backward compatibility, calls to *Index* will continue to return *Int64Index, UInt64Index* and *Float64Index* when given numeric data, but in the future, an *Index* will be returned.

**Current behavior:**

```
In [1]: pd.Index([1, 2, 3], dtype="int32")
Out [1]: Int64Index([1, 2, 3], dtype='int64')
In [2]: pd.Index([1, 2, 3], dtype="uint64")
Out [2]: UInt64Index([1, 2, 3], dtype='uint64')
```

**Future behavior:**

```
In [3]: pd.Index([1, 2, 3], dtype="int32")
Out [3]: Index([1, 2, 3], dtype='int32')
In [4]: pd.Index([1, 2, 3], dtype="uint64")
Out [4]: Index([1, 2, 3], dtype='uint64')
```

**Deprecated DataFrame.append and Series.append**

*DataFrame.append()* and *Series.append()* have been deprecated and will be removed in a future version. Use *pandas.concat()* instead (GH35407).

**Deprecated syntax**

```
In [1]: pd.Series([1, 2]).append(pd.Series([3, 4])
Out [1]:
<stdin>:1: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  0  1
  1  2
  0  3
  1  4
```

(continues on next page)
** dtype: int64  

```python
In [2]: df1 = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
In [3]: df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
In [4]: df1.append(df2)
```

```text
<stdin>:1: FutureWarning: The series.append method is deprecated and will be removed... from pandas in a future version. Use pandas.concat instead.
```

Recommended syntax

```python
In [34]: pd.concat([pd.Series([1, 2]), pd.Series([3, 4])])
```

```text
Out[34]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

dtype: int64

```python
In [35]: df1 = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
In [36]: df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
In [37]: pd.concat([df1, df2])
```

```text
Out[37]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
```

**Other Deprecations**

- Deprecated `Index.is_type_compatible()` (GH42113)
- Deprecated method argument in `Index.get_loc()`, use `index.get_indexer([label], method=...)` instead (GH42269)
- Deprecated treating integer keys in `Series.__setitem__()` as positional when the index is a `Float64Index` not containing the key, a `IntervalIndex` with no entries containing the key, or a `MultiIndex` with leading `Float64Index` level not containing the key (GH33469)
- Deprecated treating `numpy.datetime64` objects as UTC times when passed to the `Timestamp` constructor along with a timezone. In a future version, these will be treated as wall-times. To retain the old behavior, use `Timestamp(dt64).tz_localize("UTC").tz_convert(tz)` (GH24559)
- Deprecated ignoring missing labels when indexing with a sequence of labels on a level of a `MultiIndex` (GH42351)
• Creating an empty `Series` without a `dtype` will now raise a more visible `FutureWarning` instead of a `DeprecationWarning` (GH30017)

• Deprecated the `kind` argument in `Index.get_slice_bound()`, `Index.slice_indexer()`, and `Index.slice_locs()`; in a future version passing `kind` will raise (GH42857)

• Deprecated dropping of nuisance columns in `Rolling`, `Expanding`, and `EWM` aggregations (GH42738)

• Deprecated `Index.reindex()` with a non-unique `Index` (GH42568)

• Deprecated `Styler.render()` in favor of `Styler.to_html()` (GH42140)

• Deprecated `Styler.hide_index()` and `Styler.hide_columns()` in favor of `Styler.hide()` (GH43758)

• Deprecated passing in a string column label into `times` in `DataFrame.ewm()` (GH43265)

• Deprecated passing a non-unique `Index` (GH42568)

• Deprecated `Styler.hide_index()` and `Styler.hide_columns()` in favor of `Styler.hide()` (GH43758)

• Deprecated casting behavior when passing an item with mismatched-timezone to `DatetimeIndex.insert()`, `DatetimeIndex.putmask()`, `DatetimeIndex.where()` `DatetimeIndex.fillna()`, `Series.mask()`, `Series.where()`, `Series.fillna()`, `Series.shift()`, `Series.replace()`, `Series.reindex()` (and `DataFrame` column analogues). In the past this has cast to object dtype. In a future version, these will cast the passed item to the index or series's timezone (GH37605, GH44940)

• Deprecated parameter `names` in `Index.copy()` (GH44916)
- A deprecation warning is now shown for `DataFrame.to_latex()` indicating the arguments signature may change and emulate more the arguments to `Styler.to_latex()` in future versions (GH44411)
- Deprecated behavior of `concat()` between objects with bool-dtype and numeric-dtypes; in a future version these will cast to object dtypes instead of coercing bools to numeric values (GH39817)
- Deprecated `Categorical.replace()`, use `Series.replace()` instead (GH44929)
- Deprecated passing `set` or `dict` as indexer for `DataFrame.loc.__setitem__`, `DataFrame.loc.__getitem__`, `Series.loc.__setitem__`, `Series.loc.__getitem__`, `DataFrame.__getitem__`, `Series.__getitem__` and `Series.__setitem__` (GH42825)
- Deprecated `Index.__getitem__` with a bool key; use `index.values[key]` to get the old behavior (GH44051)
- Deprecated downcasting column-by-column in `DataFrame.where()` with integer-dtypes (GH44597)
- Deprecated `DatetimeIndex.union_many()`, use `DatetimeIndex.union()` instead (GH44091)
- Deprecated `Groupby.pad()` in favor of `Groupby.ffill()` (GH33396)
- Deprecated `Groupby.backfill()` in favor of `Groupby.bfill()` (GH33396)
- Deprecated `Resample.pad()` in favor of `Resample.ffill()` (GH33396)
- Deprecated `Resample.backfill()` in favor of `Resample.bfill()` (GH33396)
- Deprecated `numeric_only=None` in `DataFrame.rank()`; in a future version `numeric_only` must be either True or False (the default) (GH45036)
- Deprecated the behavior of `Timestamp.utcfromtimestamp()`, in the future it will return a timezone-aware UTC `Timestamp` (GH22451)
- Deprecated `NaT.freq()` (GH45071)
- Deprecated behavior of `Series` and `DataFrame` construction when passed float-dtype data containing NaN and an integer dtype ignoring the dtype argument; in a future version this will raise (GH40110)
- Deprecated the behaviour of `Series.to_frame()` and `Index.to_frame()` to ignore the name argument when `name=None`. Currently, this means to preserve the existing name, but in the future explicitly passing `name=None` will set None as the name of the column in the resulting DataFrame (GH44212)

**Performance improvements**

- Performance improvement in `GroupBy.sample()`, especially when weights argument provided (GH34483)
- Performance improvement when converting non-string arrays to string arrays (GH34483)
- Performance improvement in `GroupBy.transform()` for user-defined functions (GH41598)
- Performance improvement in constructing `DataFrame` objects (GH42631, GH43142, GH43147, GH43307, GH43144, GH44826)
- Performance improvement in `GroupBy.shift()` when `fill_value` argument is provided (GH26615)
- Performance improvement in `DataFrame.corr()` for `method=pearson` on data without missing values (GH40956)
- Performance improvement in some `GroupBy.apply()` operations (GH42992, GH43578)
- Performance improvement in `read_stata()` (GH43059, GH43227)
- Performance improvement in `read_sas()` (GH43333)
- Performance improvement in `to_datetime()` with uint dtypes (GH42606)
• Performance improvement in `to_datetime()` with `infer_datetime_format` set to True (GH43901)
• Performance improvement in `Series.sparse.to_coo()` (GH42880)
• Performance improvement in indexing with a `UInt64Index` (GH43862)
• Performance improvement in indexing with a `Float64Index` (GH43705)
• Performance improvement in indexing with a non-unique `Index` (GH43792)
• Performance improvement in indexing with a listlike indexer on a `MultiIndex` (GH43370)
• Performance improvement in indexing with a `MultiIndex` indexer on another `MultiIndex` (GH43370)
• Performance improvement in `GroupBy.quantile()` (GH43469, GH43725)
• Performance improvement in `GroupBy.count()` (GH43730, GH43694)
• Performance improvement in `GroupBy.any()` and `GroupBy.all()` (GH43675, GH42841)
• Performance improvement in `GroupBy.std()` (GH43115, GH43576)
• Performance improvement in `GroupBy.cumsum()` (GH43309)
• `SparseArray.min()` and `SparseArray.max()` no longer require converting to a dense array (GH43526)
• Indexing into a `SparseArray` with a slice with `step=1` no longer requires converting to a dense array (GH43777)
• Performance improvement in `SparseArray.take()` with `allow_fill=False` (GH43654)
• Performance improvement in `Rolling.mean()`, `Expanding.mean()`, `Rolling.sum()`, `Expanding.sum()`, `Rolling.max()`, `Expanding.max()`, `Rolling.min()` and `Expanding.min()` with engine="numba" (GH43612, GH44176, GH45170)
• Improved performance of `pandas.read_csv()` with `memory_map=True` when file encoding is UTF-8 (GH43787)
• Performance improvement in `RangeIndex.sort_values()` overriding `Index.sort_values()` (GH43666)
• Performance improvement in `RangeIndex.insert()` (GH43988)
• Performance improvement in `Index.insert()` (GH43953)
• Performance improvement in `DatetimeIndex.tolist()` (GH43823)
• Performance improvement in `DatetimeIndex.union()` (GH42353)
• Performance improvement in `Series.nsmallest()` (GH43696)
• Performance improvement in `DataFrame.insert()` (GH42998)
• Performance improvement in `DataFrame.dropna()` (GH43683)
• Performance improvement in `DataFrame.fillna()` (GH43316)
• Performance improvement in `DataFrame.values()` (GH43160)
• Performance improvement in `DataFrame.select_dtypes()` (GH42611)
• Performance improvement in `DataFrame` reductions (GH43185, GH43243, GH43311, GH43609)
• Performance improvement in `Series.unstack()` and `DataFrame.unstack()` (GH43335, GH43352, GH42704, GH43025)
• Performance improvement in `Series.to_frame()` (GH43558)
• Performance improvement in `Series.mad()` (GH43010)
• Performance improvement in `merge()` (GH43332)
• Performance improvement in `to_csv()` when index column is a datetime and is formatted (GH39413)
• Performance improvement in `to_csv()` when `MultiIndex` contains a lot of unused levels (GH37484)
• Performance improvement in `read_csv()` when `index_col` was set with a numeric column (GH44158)
• Performance improvement in `concat()` (GH43354)
• Performance improvement in `SparseArray.__getitem__()` (GH23122)
• Performance improvement in constructing a `DataFrame` from array-like objects like a Pytorch tensor (GH44616)

Bug fixes

Categorical

• Bug in setting dtype-incompatible values into a `Categorical` (or `Series` or `DataFrame` backed by `Categorical`) raising `ValueError` instead of `TypeError` (GH41919)
• Bug in `Categorical.searchsorted()` when passing a dtype-incompatible value raising `KeyError` instead of `TypeError` (GH41919)
• Bug in `Categorical.astype()` casting datetimes and `Timestamp` to int for dtype object (GH44930)
• Bug in `Series.where()` with `CategoricalDtype` when passing a dtype-incompatible value raising `ValueError` instead of `TypeError` (GH41919)
• Bug in `Categorical.fillna()` when passing a dtype-incompatible value raising `ValueError` instead of `TypeError` when filling with a non-category tuple (GH41919)

Datetimelike

• Bug in `DataFrame` constructor unnecessarily copying non-datetimelike 2D object arrays (GH39272)
• Bug in `to_datetime()` with `format` and `pandas.NA` was raising `ValueError` (GH42957)
• `to_datetime()` would silently swap MM/DD/YYYY and DD/MM/YYYY formats if the given `dayfirst` option could not be respected - now, a warning is raised in the case of delimited date strings (e.g. 31-12-2012) (GH12585)
• Bug in `date_range()` and `bdate_range()` do not return right bound when `start = end` and set is closed on one side (GH43394)
• Bug in inplace addition and subtraction of `DatetimeIndex` or `TimedeltaIndex` with `DatetimeArray` or `TimedeltaArray` (GH43904)
• Bug in calling `np.isnan`, `np.isfinite`, or `np.isinf` on a timezone-aware `DatetimeIndex` incorrectly raising `TypeError` (GH43917)
• Bug in constructing a `Series` from datetimelike strings with mixed timezones incorrectly partially-inferring datetime values (GH40111)
• Bug in addition of a `Tick` object and a `np.timedelta64` object incorrectly raising instead of returning `Timedelta` (GH44474)
• `np.maximum.reduce` and `np.minimum.reduce` now correctly return `Timestamp` and `Timedelta` objects when operating on `Series`, `DataFrame`, or `Index` with `datetime64[ns]` or `timedelta64[ns]` dtype (GH43923)

5.1. Version 1.4
• Bug in adding a np.timedelta64 object to a BusinessDay or CustomBusinessDay object incorrectly raising (GH44532)
• Bug in Index.insert() for inserting np.datetime64, np.timedelta64 or tuple into Index with dtype='object' with negative loc adding None and replacing existing value (GH44509)
• Bug in Timestamp.to_pydatetime() failing to retain the fold attribute (GH45087)
• Bug in Series.mode() with DatetimeTZDtype incorrectly returning timezone-naive and PeriodDtype incorrectly raising (GH41927)
• Fixed regression in reindex() raising an error when using an incompatible fill value with a datetime-like dtype (or not raising a deprecation warning for using a datetime.date as fill value) (GH42921)
• Bug in DateOffset` addition with Timestamp where offset.nanoseconds would not be included in the result (GH43968, GH36589)
• Bug in Timestamp.fromtimestamp() not supporting the tz argument (GH45083)
• Bug in DataFrame construction from dict of Series with mismatched index dtypes sometimes raising depending on the ordering of the passed dict (GH44091)
• Bug in Timestamp hashing during some DST transitions caused a segmentation fault (GH33931 and GH40817)

Timedelta

• Bug in division of all-NaT TimeDeltaIndex, Series or DataFrame column with object-dtype array like of numbers failing to infer the result as timedelta64-dtype (GH39750)
• Bug in floor division of timedelta64[ns] data with a scalar returning garbage values (GH44466)
• Bug in Timedelta now properly taking into account any nanoseconds contribution of any kwarg (GH43764, GH45227)

Time Zones

• Bug in to_datetime() with infer_datetime_format=True failing to parse zero UTC offset (Z) correctly (GH41047)
• Bug in Series.dt.tz_convert() resetting index in a Series with CategoricalIndex (GH43080)
• Bug in Timestamp and DatetimeIndex incorrectly raising a TypeError when subtracting two timezone-aware objects with mismatched timezones (GH31793)

Numeric

• Bug in floor-dividing a list or tuple of integers by a Series incorrectly raising (GH44674)
• Bug in DataFrame.rank() raising ValueError with object columns and method="first" (GH41931)
• Bug in DataFrame.rank() treating missing values and extreme values as equal (for example np.nan and np.inf), causing incorrect results when na_option="bottom" or na_option="top used (GH41931)
• Bug in numexpr engine still being used when the option compute.use_numexpr is set to False (GH32556)
• Bug in DataFrame arithmetic ops with a subclass whose _constructor() attribute is a callable other than the subclass itself (GH43201)
• Bug in arithmetic operations involving RangeIndex where the result would have the incorrect name (GH43962)
• Bug in arithmetic operations involving Series where the result could have the incorrect name when the operands having matching NA or matching tuple names (GH44459)
• Bug in division with IntegerDtype or BooleanDtype array and NA scalar incorrectly raising (GH44685)
• Bug in multiplying a Series with FloatingDtype with a timedelta-like scalar incorrectly raising (GH44772)

Conversion

• Bug in UInt64Index constructor when passing a list containing both positive integers small enough to cast to int64 and integers too large to hold in int64 (GH42201)
• Bug in Series constructor returning 0 for missing values with dtype int64 and False for dtype bool (GH43017, GH43018)
• Bug in constructing a DataFrame from a PandasArray containing Series objects behaving differently than an equivalent np.ndarray (GH43986)
• Bug in IntegerDtype not allowing coercion from string dtype (GH25472)
• Bug in to_datetime() with arg: xr.DataArray and unit="ns" specified raises TypeError (GH44053)
• Bug in DataFrame.convert_dtypes() not returning the correct type when a subclass does not overload _constructor_sliced() (GH43201)
• Bug in DataFrame.astype() not propagating attrs from the original DataFrame (GH44414)
• Bug in DataFrame.convert_dtypes() result losing columns.names (GH41435)
• Bug in constructing a IntegerArray from pyarrow data failing to validate dtypes (GH44891)
• Bug in Series.astype() not allowing converting from a PeriodDtype to datetime64 dtype, inconsistent with the PeriodIndex behavior (GH45038)

Strings

• Bug in checking for string[pyarrow] dtype incorrectly raising an ImportError when pyarrow is not installed (GH44276)

Interval

• Bug in Series.where() with IntervalDtype incorrectly raising when the where call should not replace anything (GH44181)

Indexing

• Bug in Series.rename() with MultiIndex and level is provided (GH43659)
• Bug in DataFrame.truncate() and Series.truncate() when the object's Index has a length greater than one but only one unique value (GH42365)
• Bug in Series.loc() and DataFrame.loc() with a MultiIndex when indexing with a tuple in which one of the levels is also a tuple (GH27591)
• Bug in Series.loc() with a MultiIndex whose first level contains only np.nan values (GH42055)
• Bug in indexing on a Series or DataFrame with a DatetimeIndex when passing a string, the return type depended on whether the index was monotonic (GH24892)
• Bug in indexing on a `MultiIndex` failing to drop scalar levels when the indexer is a tuple containing a datetime-like string (GH42476)

• Bug in `DataFrame.sort_values()` and `Series.sort_values()` when passing an ascending value, failed to raise or incorrectly raising `ValueError` (GH41634)

• Bug in updating values of `pandas.Series` using boolean index, created by using `pandas.DataFrame.pop()` (GH42530)

• Bug in `Index.get_indexer_non_unique()` when index contains multiple `np.nan` (GH35392)

• Bug in `DataFrame.query()` did not handle the degree sign in a backticked column name, such as `"Temp(°C)"`, used in an expression to query a `DataFrame` (GH42826)

• Bug in `DataFrame.drop()` where the error message did not show missing labels with commas when raising `KeyError` (GH42881)

• Bug in `DataFrame.query()` where method calls in query strings led to errors when the `numexpr` package was installed (GH22435)

• Bug in `DataFrame.nlargest()` and `Series.nlargest()` where sorted result did not count indexes containing `np.nan` (GH28984)

• Bug in indexing on a non-unique object-dtype `Index` with an NA scalar (e.g. `np.nan`) (GH43711)

• Bug in `DataFrame.__setitem__()` incorrectly writing into an existing column’s array rather than setting a new array when the new dtype and the old dtype match (GH43406)

• Bug in setting floating-dtype values into a `Series` with integer dtype failing to set inplace when those values can be losslessly converted to integers (GH43166)

• Bug in `Series.__setitem__()` with object dtype when setting an array with matching size and dtype='datetime64[ns]' or dtype='timedelta64[ns]' incorrectly converting the datetime/timedeltas to integers (GH43868)

• Bug in `DataFrame.drop()` where `ignore_index=True` was not being respected when the index was already sorted (GH43591)

• Bug in `Index.get_indexer_non_unique()` when index contains multiple `np.datetime64("NaT")` and `np.timedelta64("NaT")` (GH43869)

• Bug in indexing a non-unique object-dtype `Index` with an NA scalar (e.g. `np.nan`) (GH44201)

• Bug in `Series.__setitem__()` with an integer dtype other than `int64` setting with a `range` object unnecessarily upcasting to `int64` (GH44261)

• Bug in `Series.__setitem__()` with a boolean mask indexer setting a listlike value of length 1 incorrectly broadcasting that value (GH44265)

• Bug in `Series.reset_index()` not ignoring name argument when drop and inplace are set to `True` (GH44575)

• Bug in `DataFrame.loc.__setitem__()` and `DataFrame.iloc.__setitem__()` with mixed dtypes sometimes failing to operate in-place (GH44345)

• Bug in `DataFrame.loc.__getitem__()` incorrectly raising `KeyError` when selecting a single column with a boolean key (GH44322).

• Bug in `Series.to_datetime()` with a single `ExtensionDtype` column and setting 2D values e.g. `df.iloc[:,] = df.values` incorrectly raising (GH44514)
• Bug in setting values with `DataFrame.iloc()` with a single `ExtensionDtype` column and a tuple of arrays as the indexer (GH44703)

• Bug in indexing on columns with `loc` or `iloc` using a slice with a negative step with `ExtensionDtype` columns incorrectly raising (GH44551)

• Bug in `DataFrame.loc.__setitem__()` changing dtype when indexer was completely False (GH37550)

• Bug in `IntervalIndex.get_indexer_non_unique()` returning boolean mask instead of array of integers for a non unique and non monotonic index (GH44084)

• Bug in `IntervalIndex.get_indexer_non_unique()` not handling targets of dtype 'object' with NaNs correctly (GH44482)

• Fixed regression where a single column `np.matrix` was no longer coerced to a 1d `np.ndarray` when added to a `DataFrame` (GH42376)

• Bug in `Series.__getitem__()` with a `CategoricalIndex` of integers treating lists of integers as positional indexers, inconsistent with the behavior with a single scalar integer (GH15470, GH14865)

• Bug in `Series.__setitem__()` when setting floats or integers into integer-dtype `Series` failing to upcast when necessary to retain precision (GH45121)

• Bug in `DataFrame.iloc.__setitem__()` ignores axis argument (GH45032)

**Missing**

• Bug in `DataFrame.fillna()` with `limit` and no method ignores axis='columns' or axis = 1 (GH40989, GH17399)

• Bug in `DataFrame.fillna()` not replacing missing values when using a dict-like value and duplicate column names (GH43476)

• Bug in constructing a `DataFrame` with a dictionary `np.datetime64` as a value and `dtype='timedelta64[ns]'`, or vice-versa, incorrectly casting instead of raising (GH44428)

• Bug in `Series.interpolate()` and `DataFrame.interpolate()` with `inplace=True` not writing to the underlying array(s) in-place (GH44749)

• Bug in `Index.fillna()` incorrectly returning an unfilled `Index` when NA values are present and `downcast` argument is specified. This now raises `NotImplementedError` instead; do not pass `downcast` argument (GH44473)

• Bug in `DataFrame.dropna()` changing `Index` even if no entries were dropped (GH41965)

• Bug in `Series.fillna()` with an object-dtype incorrectly ignoring `downcast="infer"` (GH44241)

**MultiIndex**

• Bug in `MultiIndex.get_loc()` where the first level is a `DatetimeIndex` and a string key is passed (GH42465)

• Bug in `MultiIndex.reindex()` when passing a level that corresponds to an `ExtensionDtype` level (GH42043)

• Bug in `MultiIndex.get_loc()` raising `TypeError` instead of `KeyError` on nested tuple (GH42440)

• Bug in `MultiIndex.union()` setting wrong `sortorder` causing errors in subsequent indexing operations with slices (GH44752)

• Bug in `MultiIndex.putmask()` where the other value was also a `MultiIndex` (GH43212)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in `MultiIndex.dtypes()` duplicate level names returned only one dtype per name (GH45174)

I/O

- Bug in `read_excel()` attempting to read chart sheets from .xlsx files (GH41448)
- Bug in `json_normalize()` where `errors=ignore` could fail to ignore missing values of `meta` when `record_path` has a length greater than one (GH41876)
- Bug in `read_csv()` with multi-header input and arguments referencing column names as tuples (GH42446)
- Bug in `read_fwf()`, where difference in lengths of `colspecs` and `names` was not raising `ValueError` (GH40830)
- Bug in `Series.to_json()` and `DataFrame.to_json()` where some attributes were skipped when serializing plain Python objects to JSON (GH42768, GH33043)
- Column headers are dropped when constructing a `DataFrame` from a sqlalchemy's `Row` object (GH40682)
- Bug in unpickling an `Index` with object dtype incorrectly inferring numeric dtypes (GH43188)
- Bug in `read_csv()` where reading multi-header input with unequal lengths incorrectly raised `IndexError` (GH43102)
- Bug in `read_csv()` raising `ParserError` when reading file in chunks and some chunk blocks have fewer columns than header for `engine="c"` (GH21211)
- Bug in `read_csv()`, changed exception class when expecting a file path name or file-like object from `OSError` to `TypeError` (GH43366)
- Bug in `json_normalize()` where multi-character `sep` parameter is incorrectly prefixed to every key (GH43831)
- Bug in `json_normalize()` where reading data with missing multi-level metadata would not respect `errors="ignore"` (GH44312)
- Bug in `read_csv()` used second row to guess implicit index if `header` was set to `None` for `engine="python"` (GH22144)
- Bug in `read_csv()` not recognizing bad lines when `names` were given for `engine="c"` (GH22144)
- Bug in `read_csv()` with `float_precision="round_trip"` which did not skip initial/trailing whitespace (GH43713)
- Bug when Python is built without the lzma module: a warning was raised at the pandas import time, even if the lzma capability isn’t used (GH43495)
- Bug in `read_csv()` not applying dtype for `index_col` (GH9435)
- Bug in dumping/loading a `DataFrame` with `yaml.dump(frame)` (GH42748)
- Bug in `read_csv()` raising `ValueError` when `names` was longer than `header` but equal to data rows for `engine="python"` (GH38453)
- Bug in `ExcelWriter`, where `engine_kwags` were not passed through to all engines (GH43442)
- Bug in `read_csv()` raising `ValueError` when `parse_dates` was used with `MultiIndex` columns (GH8991)
• Bug in `read_csv()` not raising a `ValueError` when `\n` was specified as `delimiter` or `sep` which conflicts with `lineterminator` (GH43528)

• Bug in `to_csv()` converting datetimes in categorical `Series` to integers (GH40754)

• Bug in `read_csv()` converting columns to numeric after date parsing failed (GH11019)

• Bug in `read_csv()` not replacing `NaN` values with `np.nan` before attempting date conversion (GH26203)

• Bug in `read_csv()` raising `AttributeError` when attempting to read a `.csv` file and infer index column dtype from an nullable integer type (GH44079)

• Bug in `to_csv()` always coercing datetime columns with different formats to the same format (GH21734)

• `DataFrame.to_csv()` and `Series.to_csv()` with `compression` set to 'zip' no longer create a zip file containing a file ending with “.zip”. Instead, they try to infer the inner file name more smartly (GH39465)

• Bug in `read_csv()` where reading a mixed column of booleans and missing values to a float type results in the missing values becoming 1.0 rather than NaN (GH42808, GH34120)

• Bug in `to_csv()` raising error for `pd.NA` with extension array dtype (GH43903)

• Bug in `read_csv()` when passing simultaneously a parser in `date_parser` and `parse_dates=False`, the parsing was still called (GH44366)

• Bug in `read_csv()` not setting name of `MultiIndex` columns correctly when `index_col` is not the first column (GH38549)

• Bug in `read_csv()` silently ignoring errors when failing to create a memory-mapped file (GH44766)

• Bug in `read_csv()` when passing a `tempfile.SpooledTemporaryFile` opened in binary mode (GH44748)

• Bug in `read_json()` raising `ValueError` when attempting to parse json strings containing “://” (GH36271)

• Bug in `read_csv()` when engine="c" and `encoding_errors=None` which caused a segfault (GH45180)

• Bug in `read_csv()` an invalid value of `usecols` leading to an unclosed file handle (GH45384)

• Bug in `DataFrame.to_json()` fix memory leak (GH43877)

**Period**

• Bug in adding a `Period` object to a `np.timedelta64` object incorrectly raising `TypeError` (GH44182)

• Bug in `PeriodIndex.to_timestamp()` when the index has `freq="B"` inferring `freq="D"` for its result instead of `freq="B"` (GH44105)

• Bug in `Period` constructor incorrectly allowing `np.timedelta64("NaT")` (GH44507)

• Bug in `PeriodIndex.to_timestamp()` giving incorrect values for indexes with non-contiguous data (GH44100)

• Bug in `Series.where()` with `PeriodDtype` incorrectly raising when the `where` call should not replace anything (GH45135)
Plotting

- When given non-numeric data, DataFrame.boxplot() now raises a ValueError rather than a cryptic KeyError or ZeroDivisionError, in line with other plotting functions like DataFrame.hist() (GH43480)

Groupby/resample/rolling

- Bug in SeriesGroupBy.apply() where passing an unrecognized string argument failed to raise TypeError when the underlying Series is empty (GH42021)
- Bug in Series.rolling.apply(), DataFrame.rolling.apply(), Series.expanding.apply() and DataFrame.expanding.apply() with engine="numba" where *args were being cached with the user passed function (GH42287)
- Bug in GroupBy.max() and GroupBy.min() with nullable integer dtypes losing precision (GH41743)
- Bug in DataFrame.groupby.rolling.var() would calculate the rolling variance only on the first group (GH42442)
- Bug in GroupBy.shift() that would return the grouping columns if fill_value was not None (GH41556)
- Bug in SeriesGroupBy.nlargest() and SeriesGroupBy.nsmallest() would have an inconsistent index when the input Series was sorted and n was greater than or equal to all group sizes (GH15272, GH16345, GH29129)
- Bug in pandas.DataFrame.ewm(), where non-float64 dtypes were silently failing (GH42452)
- Bug in pandas.DataFrame.rolling() operation along rows (axis=1) incorrectly omits columns containing float16 and float32 (GH41779)
- Bug in Resampler.aggregate() did not allow the use of Named Aggregation (GH32803)
- Bug in Series.rolling() when the Series dtype was Int64 (GH43016)
- Bug in DataFrame.rolling.corr() when the DataFrame columns was a MultiIndex (GH21157)
- Bug in DataFrame.groupby.rolling() when specifying on and calling __getitem__ would subsequently return incorrect results (GH43355)
- Bug in GroupBy.apply() with time-based Grouper objects incorrectly raising ValueError in corner cases where the grouping vector contains a NaT (GH43500, GH43515)
- Bug in GroupBy.mean() failing with complex dtype (GH43701)
- Bug in Series.rolling() and DataFrame.rolling() for centered datetime-like windows with uneven nanosecond (GH43997)
- Bug in GroupBy.mean() raising KeyError when column was selected at least twice (GH44924)
- Bug in GroupBy.nth() failing on axis=1 (GH43926)
- Bug in Series.rolling() and DataFrame.rolling() not respecting right bound on centered datetime-like windows, if the index contain duplicates (GH3944)
- Bug in Series.rolling() and DataFrame.rolling() when using a pandas.api.indexers. BaseIndexer subclass that returned unequal start and end arrays would segfault instead of raising a ValueError (GH44470)
- Bug in Groupby.nunique() not respecting observed=True for categorical grouping columns (GH45128)
• Bug in GroupBy.head() and GroupBy.tail() not dropping groups with NaN when dropna=True (GH45089)
• Bug in GroupBy.__iter__() after selecting a subset of columns in a GroupBy object, which returned all columns instead of the chosen subset (GH44821)
• Bug in Groupby.rolling() when non-monotonic data passed, fails to correctly raise ValueError (GH43909)
• Bug where grouping by a Series that has a categorical data type and length unequal to the axis of grouping raised ValueError (GH44179)

Reshaping

• Improved error message when creating a DataFrame column from a multi-dimensional numpy.ndarray (GH42463)
• Bug in concat() creating MultiIndex with duplicate level entries when concatenating a DataFrame with duplicates in Index and multiple keys (GH42651)
• Bug in pandas.cut() on Series with duplicate indices and non-exact pandas.CategoricalIndex() (GH42185, GH42425)
• Bug in DataFrame.append() failing to retain dtypes when appended columns do not match (GH43392)
• Bug in concat() of bool and boolean dtypes resulting in object dtype instead of boolean dtype (GH42800)
• Bug in crosstab() when inputs are categorical Series, there are categories that are not present in one or both of the Series, and margins=True. Previously the margin value for missing categories was NaN. It is now correctly reported as 0 (GH43505)
• Bug in concat() would fail when the objs argument all had the same index and the keys argument contained duplicates (GH43595)
• Bug in concat() which ignored the sort parameter (GH43375)
• Bug in merge() with MultiIndex as column index for the on argument returning an error when assigning a column internally (GH43734)
• Bug in crosstab() would fail when inputs are lists or tuples (GH44076)
• Bug in DataFrame.append() failing to retain index.name when appending a list of Series objects (GH44109)
• Fixed metadata propagation in Dataframe.apply() method, consequently fixing the same issue for Dataframe.transform(), Dataframe.nunique() and Dataframe.mode() (GH28283)
• Bug in concat() casting levels of MultiIndex to float if all levels only consist of missing values (GH44900)
• Bug in DataFrame.stack() with ExtensionDtype columns incorrectly raising (GH43561)
• Bug in merge() raising KeyError when joining over differently named indexes with on keywords (GH45094)
• Bug in Series.unstack() with object doing unwanted type inference on resulting columns (GH44595)
• Bug in MultiIndex.join() with overlapping IntervalIndex levels (GH44096)
• Bug in DataFrame.replace() and Series.replace() results is different dtype based on regex parameter (GH44864)
• Bug in DataFrame.pivot() with index=None when the DataFrame index was a MultiIndex (GH23955)
Sparse

- Bug in `DataFrame.sparse.to_coo()` raising `AttributeError` when column names are not unique (GH29564)
- Bug in `SparseArray.max()` and `SparseArray.min()` raising `ValueError` for arrays with 0 non-null elements (GH43527)
- Bug in `DataFrame.sparse.to_coo()` silently converting non-zero fill values to zero (GH24817)
- Bug in `SparseArray` comparison methods with an array-like operand of mismatched length raising `AssertionError` or unclear `ValueError` depending on the input (GH43863)
- Bug in `SparseArray` arithmetic methods `floordiv` and `mod` behaviors when dividing by zero not matching the non-sparse `Series` behavior (GH38172)
- Bug in `SparseArray` unary methods as well as `SparseArray.isna()` doesn't recalculate indexes (GH44955)

ExtensionArray

- Bug in `array()` failing to preserve `PandasArray` (GH43887)
- NumPy ufuncs `np.abs`, `np.positive`, `np.negative` now correctly preserve dtype when called on ExtensionArrays that implement `__abs__`, `__pos__`, `__neg__`, respectively. In particular this is fixed for `TimedeltaArray` (GH43899, GH23316)
- NumPy ufuncs `np.minimum.reduce`, `np.maximum.reduce`, `np.add.reduce`, and `np.prod.reduce` now work correctly instead of raising `NotImplementedError` on `Series` with `IntegerDtype` or `FloatDtype` (GH43923, GH44793)
- NumPy ufuncs with `out` keyword are now supported by arrays with `IntegerDtype` and `FloatingDtype` (GH45122)
- Avoid raising `PerformanceWarning` about fragmented `DataFrame` when using many columns with an extension dtype (GH44098)
- Bug in `IntegerArray` and `FloatingArray` construction incorrectly coercing mismatched NA values (e.g. `np.timedelta64("NaT")`) to numeric NA (GH44514)
- Bug in `BooleanArray.__eq__()` and `BooleanArray.__ne__()` raising `TypeError` on comparison with an incompatible type (like a string). This caused `DataFrame.replace()` to sometimes raise a `TypeError` if a nullable boolean column was included (GH44499)
- Bug in `array()` incorrectly raising when passed a `ndarray` with `float16` dtype (GH44715)
- Bug in calling `np.sqrt` on `BooleanArray` returning a malformed `FloatingArray` (GH44715)
- Bug in `Series.where()` with `ExtensionDtype` when other is a NA scalar incompatible with the `Series` dtype (e.g. `NaT` with a numeric dtype) incorrectly casting to a compatible NA value (GH44697)
- Bug in `Series.replace()` where explicitly passing `value=None` is treated as if no value was passed, and `None` not being in the result (GH36984, GH19998)
- Bug in `Series.replace()` with unwanted downcasting being done in no-op replacements (GH44498)
- Bug in `Series.replace()` with `FloatDtype`, `string[python]`, or `string[pyarrow]` dtype not being preserved when possible (GH33484, GH40732, GH31644, GH41215, GH25438)
Styler

- Bug in `Styler` where the `uuid` at initialization maintained a floating underscore (GH43037)
- Bug in `Styler.to_html()` where the `Styler` object was updated if the `to_html` method was called with some args (GH43034)
- Bug in `Styler.copy()` where `uuid` was not previously copied (GH40675)
- Bug in `Styler.apply()` where functions which returned `Series` objects were not correctly handled in terms of aligning their index labels (GH13657, GH42014)
- Bug when rendering an empty `DataFrame` with a named `Index` (GH43305)
- Bug when rendering a single level `MultiIndex` (GH43383)
- Bug when combining non-sparse rendering and `Styler.hide_columns()` or `Styler.hide_index()` (GH43464)
- Bug setting a table style when using multiple selectors in `Styler` (GH44011)
- Bugs where row trimming and column trimming failed to reflect hidden rows (GH43703, GH44247)

Other

- Bug in `DataFrame.astype()` with non-unique columns and a `Series` `dtype` argument (GH44417)
- Bug in `CustomBusinessMonthBegin.__add__()` (`CustomBusinessMonthEnd.__add__()`) not applying the extra `offset` parameter when beginning (end) of the target month is already a business day (GH41356)
- Bug in `RangeIndex.union()` with another `RangeIndex` with matching (even) `step` and starts differing by strictly less than `step / 2` (GH44019)
- Bug in `RangeIndex.difference()` with `sort=None` and `step<0` failing to sort (GH44085)
- Bug in `Series.replace()` and `DataFrame.replace()` with `value=None` and ExtensionDtypes (GH44270, GH37899)
- Bug in `FloatingArray.equals()` failing to consider two arrays equal if they contain `np.nan` values (GH44382)
- Bug in `DataFrame.shift()` with `axis=1` and ExtensionDtype columns incorrectly raising when an incompatible `fill_value` is passed (GH44564)
- Bug in `DataFrame.shift()` with `axis=1` and `periods` larger than `len(frame.columns)` producing an invalid `DataFrame` (GH44978)
- Bug in `DataFrame.diff()` when passing a NumPy integer object instead of an `int` object (GH44572)
- Bug in `Series.replace()` raising `ValueError` when using `regex=True` with a `Series` containing `np.nan` values (GH34344)
- Bug in `DataFrame.to_records()` where an incorrect `n` was used when missing names were replaced by `level_n` (GH44818)
- Bug in `DataFrame.eval()` where `resolvers` argument was overriding the default resolvers (GH34966)
- `Series.__repr__()` and `DataFrame.__repr__()` no longer replace all null-values in indexes with “NaN” but use their real string-representations. “NaN” is used only for `float("nan")` (GH45263)
Contributors

A total of 275 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Abhishek R
- Albert Villanova del Moral
- Alessandro Bisiani +
- Alex Lim
- Alex-Gregory-1 +
- Alexander Gorodetsky
- Alexander Regueiro +
- Alexey Györi
- Alexis Mignon
- Aleš Erjavec
- Ali McMaster
- Alibi +
- Andrei Batomunkuev +
- Andrew Eckart +
- Andrew Hawyrluk
- Andrew Wood
- Anton Lodder +
- Armin Berres +
- Arushi Sharma +
- Benedikt Heidrich +
- Beni Bienz +
- Benoît Vinot
- Bert Palm +
- Boris Rumyantsev +
- Brian Hulette
- Brock
- Bruno Costa +
- Bryan Racic +
- Caleb Epstein
- Calvin Ho
- Christof Kaufmann +
- Christopher Yeh +
- Chuliang Xiao +
• ClaudiaSilver +
• DSM
• Daniel Coll +
• Daniel Schmidt +
• Dare Adewumi
• David +
• David Sanders +
• David Wales +
• Derzan Chiang +
• DeviousLab +
• Dhruv B Shetty +
• Digres45 +
• Dominik Kutra +
• Drew Levitt +
• DriesS
• EdAbati
• Elle
• Elliot Rampono
• Endre Mark Borza
• Erfan Nariman
• Evgeny Naumov +
• Ewout ter Hoeven +
• Fangchen Li
• Felix Divo
• Felix Dulys +
• Francesco Andreuzzi +
• Francois Dion +
• Frans Larsson +
• Fred Reiss
• GYvan
• Gabriel Di Pardi Arruda +
• Gesa Stupperich
• Giacomo Caria +
• Greg Siano +
• Griffin Ansel
• Hiroaki Ogasawara +
• Horace +
• Horace Lai +
• Irv Lustig
• Isaac Virshup
• JHM Darbyshire (MBP)
• JHM Darbyshire (iMac)
• JHM Darbyshire +
• Jack Liu
• Jacob Skwirsk +
• Jaime Di Cristina +
• James Holcombe +
• Janosh Riebesell +
• Jarrod Millman
• Jason Bian +
• Jeff Reback
• Jernej Makovsek +
• Jim Bradley +
• Joel Gibson +
• Joerdefloep +
• Johannes Mueller +
• John S Bogaardt +
• John Zangwill +
• Jon Haitz Legarreta Gorroño +
• Jon Wiggins +
• Jonas Haag +
• Joris Van den Bossche
• Josh Friedlander
• José Duarte +
• Julian Fleischer +
• Julien de la Bruère-T
• Justin McOmie
• Kadatatlu Kishore +
• Kaiqi Dong
• Kashif Khan +
• Kavya9986 +
• Kendall +
• Kevin Sheppard
• Kiley Hewitt
• Koen Roelofs +
• Krishna Chivukula
• KrishnaSai2020
• Leonardo Freua +
• Leonardus Chen
• Liang-Chi Hsieh +
• Loic Diridolllou +
• Lorenzo Maffioli +
• Luke Manley +
• LunarLanding +
• Marc Garcia
• Marcel Bittar +
• Marcel Gerber +
• Marco Edward Gorelli
• Marco Gorelli
• MarcoGorelli
• Marvin +
• Mateusz Piotrowski +
• Mathias Hauser +
• Matt Richards +
• Matthew Davis +
• Matthew Roeschke
• Matthew Zeitlin
• Matthias Bussonnier
• Matti Picus
• Mauro Silberberg +
• Maxim Ivanov
• Maximilian Carr +
• MesseeeksMachine
• Michael Sarrazin +
• Michael Wang +
• Michal Górny +
• Mike Phung +
• Mike Taves +
• Mohamad Hussein Rkein +
• NJOKU OKECHUKWU VALENTINE +
• Neal McBurnett +
• Nick Anderson +
• Nikita Sobolev +
• Olivier Cavadenti +
• PApostol +
• Pandas Development Team
• Patrick Hoeffler
• Peter
• Peter Tillmann +
• Prabha Arivalagan +
• Pradyumna Rahul
• Prerana Chakraborty
• Prithvijit +
• Rahul Gaikwad +
• Ray Bell
• Ricardo Martins +
• Richard Shadrach
• Robbert-jan ‘t Hoen +
• Robert Voyer +
• Robin Raymond +
• Rohan Sharma +
• Rohan Sirohia +
• Roman Yurchak
• Ruan Pretorius +
• Sam James +
• Scott Talbert
• Shashwat Sharma +
• Sheogorath27 +
• Shiv Gupta
• Shoham Debnath
• Simon Hawkins
• Soumya +
• Stan West +
• Stefanie Molin +
• Stefano Alberto Russo +
• Stephan Heßelmann
• Stephen
• Suyash Gupta +
• Sven
• Swanand01 +
• Sylvain Marié +
• TLouf
• Tania Allard +
• Terji Petersen
• TheDerivator +
• Thomas Dickson
• Thomas Kastl +
• Thomas Kluyver
• Thomas Li
• Thomas Smith
• Tim Swast
• Tim Tran +
• Tobias McNulty +
• Tobias Pitters
• Tomoki Nakagawa +
• Tony Hirst +
• Torsten Wörtwein
• V.I. Wood +
• Vaibhav K +
• Valentin Oliver Loftsson +
• Varun Shrivastava +
• Vivek Thazhathattil +
• Vyom Pathak
• Wenjun Si
• William Andrea +
• William Bradley +
• Wojciech Sadowski +
• Yao-Ching Huang +
• Yash Gupta +
• Yiannis Hadjicharalambous +
• Yoshiki Vázquez Baeza
• Yuanhao Geng
• Yury Mikhailov
• Yvan Gatete +
• Yves Delley +
• Zach Rait
• Zbyszek Królikowski +
• Zero +
• Zheyuan
• Zhiyi Wu +
• aiudirog
• ali sayyah +
• aneesh98 +
• aptalca
• arw2019 +
• attack68
• brendandrury +
• bubblingoak +
• calvinsomething +
• claws +
• deponovo +
• dicristina
• el-g-l +
• evensure +
• fotino21 +
• fshi01 +
• gfkang +
• github-actions[bot]
• i-aki-y
• jbrockmendel
• jreback
• juliandwain +
• jxb4892 +
• kendall smith +
• lmcindewar +
• lrepiton
5.2 Version 1.3

5.2.1 What’s new in 1.3.5 (December 12, 2021)

These are the changes in pandas 1.3.5. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Fixed regression in `Series.equals()` when comparing floats with dtype object to None (GH44190)
- Fixed regression in `merge_asof()` raising error when array was supplied as join key (GH42844)
- Fixed regression when resampling `DataFrame` with `DateTimeIndex` with empty groups and `uint8`, `uint16` or `uint32` columns incorrectly raising `RuntimeError` (GH43329)
- Fixed regression in creating a `DataFrame` from a timezone-aware `Timestamp` scalar near a Daylight Savings Time transition (GH42505)
- Fixed performance regression in `read_csv()` (GH44106)
- Fixed regression in `Series.duplicated()` and `Series.drop_duplicates()` when Series has `Categorical` dtype with boolean categories (GH44351)
- Fixed regression in `GroupBy.sum()` with `timedelta64[ns]` dtype containing `NaT` failing to treat that value as NA (GH42659)
- Fixed regression in `RollingGroupby.cov()` and `RollingGroupby.corr()` when other had the same shape as each group would incorrectly return superfluous groups in the result (GH42915)

Contributors

A total of 10 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Ali McMaster
- Matthew Roeschke
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Simon Hawkins
- Thomas Li
- Tobias Pitters
- jbrockmendel

5.2.2 What’s new in 1.3.4 (October 17, 2021)

These are the changes in pandas 1.3.4. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Fixed regression in `DataFrame.convert_dtypes()` incorrectly converts byte strings to strings (GH43183)
- Fixed regression in `GroupBy.agg()` where it was failing silently with mixed data types along axis=1 and `MultiIndex` (GH43209)
- Fixed regression in `merge()` with integer and NaN keys failing with outer merge (GH43550)
- Fixed regression in `DataFrame.corr()` raising `ValueError` with method="spearman" on 32-bit platforms (GH43588)
- Fixed performance regression in `MultiIndex.equals()` (GH43549)
- Fixed performance regression in `GroupBy.first()` and `GroupBy.last()` with `StringDtype` (GH41596)
- Fixed regression in `Series.cat.reorder_categories()` failing to update the categories on the Series (GH43334)
- Fixed regression in `Series.cat.categories()` setter failing to update the categories on the Series (GH43334)
- Fixed regression in `read_csv()` raising `UnicodeDecodeError` exception when `memory_map=True` (GH43540)
- Fixed regression in `DataFrame.explode()` raising `AssertionError` when column is any scalar which is not a string (GH43314)
- Fixed regression in `Series.aggregate()` attempting to pass `args` and `kwargs` multiple times to the user supplied `func` in certain cases (GH43357)
- Fixed regression when iterating over a `DataFrame.groupby.rolling` object causing the resulting DataFrames to have an incorrect index if the input groupings were not sorted (GH43386)
- Fixed regression in `DataFrame.groupby.rolling.cov()` and `DataFrame.groupby.rolling.corr()` computing incorrect results if the input groupings were not sorted (GH43386)

Bug fixes

- Fixed bug in `pandas.DataFrame.groupby.rolling()` and `pandas.api.indexers.FixedForwardWindowIndexer` leading to segfaults and window endpoints being mixed across groups (GH43267)
- Fixed bug in `GroupBy.mean()` with datetimelike values including NaT values returning incorrect results (GH43132)
- Fixed bug in `Series.aggregate()` not passing the first `args` to the user supplied `func` in certain cases (GH43357)
- Fixed memory leaks in `Series.rolling.quantile()` and `Series.rolling.median()` (GH43339)
Other

- The minimum version of Cython needed to compile pandas is now 0.29.24 (GH43729)

Contributors

A total of 17 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Alexey Györi +
- DSM
- Irv Lustig
- Jeff Reback
- Julien de la Bruère-T +
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Richard Shadrach
- Shoham Debnath
- Simon Hawkins
- Thomas Li
- aptalca +
- jbrockmendel
- michal-gh +
- realead

5.2.3 What’s new in 1.3.3 (September 12, 2021)

These are the changes in pandas 1.3.3. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in DataFrame constructor failing to broadcast for defined Index and len one list of Timestamp (GH42810)
- Fixed regression in GroupBy.agg() incorrectly raising in some cases (GH42390)
- Fixed regression in GroupBy.apply() where nan values were dropped even with dropna=False (GH43205)
- Fixed regression in GroupBy.quantile() which was failing with pandas.NA (GH42849)
- Fixed regression in merge() where on columns with ExtensionDtype or bool data types were cast to object in right and outer merge (GH40073)
- Fixed regression in RangeIndex.where() and RangeIndex.putmask() raising AssertionError when result did not represent a RangeIndex (GH43240)
• Fixed regression in `read_parquet()` where the `fastparquet` engine would not work properly with fastparquet 0.7.0 (GH43075)
• Fixed regression in `DataFrame.loc.__setitem__()` raising `ValueError` when setting array as cell value (GH43422)
• Fixed regression in `is_list_like()` where objects with `__iter__` set to None would be identified as iterable (GH43373)
• Fixed regression in `DataFrame.__getitem__()` raising error for slice of `DatetimeIndex` when index is non-monotonic (GH43223)
• Fixed regression in `Resampler.aggregate()` when used after column selection would raise if `func` is a list of aggregation functions (GH42905)
• Fixed regression in `DataFrame.corr()` where Kendall correlation would produce incorrect results for columns with repeated values (GH43401)
• Fixed regression in `DataFrame.groupby()` where aggregation on columns with object types dropped results on those columns (GH42395, GH43108)
• Fixed regression in `Series.fillna()` raising `TypeError` when filling `float Series` with list-like fill value having a dtype which couldn’t cast lostlessly (like `float32` filled with `float64`) (GH43424)
• Fixed regression in `read_csv()` raising `AttributeError` when the file handle is an `tempfile.SpooledTemporaryFile` object (GH43439)
• Fixed performance regression in `core.window.ewm.ExponentialMovingWindow.mean()` (GH42333)

**Performance improvements**

• Performance improvement for `DataFrame.__setitem__()` when the key or value is not a `DataFrame`, or key is not list-like (GH43274)

**Bug fixes**

• Fixed bug in `DataFrameGroupBy.agg()` and `DataFrameGroupBy.transform()` with `engine="numba"` where index data was not being correctly passed into `func` (GH43133)

**Contributors**

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Ali McMaster
• Irv Lustig
• Matthew Roeschke
• Matthew Zeitlin
• MeeseeksMachine
• Pandas Development Team
• Patrick Hoefler
• Prerana Chakraborty +
• Richard Shadrach
5.2.4 What’s new in 1.3.2 (August 15, 2021)

These are the changes in pandas 1.3.2. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Performance regression in `DataFrame.isin()` and `Series.isin()` for nullable data types (GH42714)
- Regression in updating values of `Series` using boolean index, created by using `DataFrame.pop()` (GH42530)
- Regression in `DataFrame.from_records()` with empty records (GH42456)
- Fixed regression in `DataFrame.shift()` where `TypeError` occurred when shifting DataFrame created by concatenation of slices and fills with values (GH42719)
- Regression in `DataFrame.agg()` when the `func` argument returned lists and `axis=1` (GH42727)
- Regression in `DataFrame.drop()` does nothing if `MultiIndex` has duplicates and indexer is a tuple or list of tuples (GH42771)
- Fixed regression where `read_csv()` raised a `ValueError` when parameters `names` and `prefix` were both set to `None` (GH42387)
- Fixed regression in comparisons between `Timestamp` object and `datetime64` objects outside the implementation bounds for nanosecond `datetime64` (GH42794)
- Fixed regression in `Styler.highlight_min()` and `Styler.highlight_max()` where `pandas.NA` was not successfully ignored (GH42650)
- Fixed regression in `concat()` where `copy=False` was not honored in `axis=1` Series concatenation (GH42501)
- Regression in `Series.nlargest()` and `Series.nsmallest()` with nullable integer or float dtype (GH42816)
- Fixed regression in `Series.quantile()` with `Int64Dtype` (GH42626)
- Fixed regression in `Series.groupby()` and `DataFrame.groupby()` where supplying the `by` argument with a Series named with a tuple would incorrectly raise (GH42731)
Bug fixes

- Bug in `read_excel()` modifies the dtypes dictionary when reading a file with duplicate columns (GH42462)
- 1D slices over extension types turn into N-dimensional slices over ExtensionArrays (GH42430)
- Fixed bug in `Series.rolling()` and `DataFrame.rolling()` not calculating window bounds correctly for the first row when `center=True` and `window` is an offset that covers all the rows (GH42753)
- `Styler.hide_columns()` now hides the index name header row as well as column headers (GH42101)
- `Styler.set_sticky()` has amended CSS to control the column/index names and ensure the correct sticky positions (GH42537)
- Bug in de-serializing datetime indexes in PYTHONOPTIMIZED mode (GH42866)

Contributors

A total of 16 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Alexander Gorodetsky +
- Fangchen Li
- Fred Reiss
- Justin McOmie +
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Richard Shadrach
- Shoham Debnath
- Simon Hawkins
- Thomas Li
- Wenjun Si
- attack68
- dicristina +
- jbrockmendel

5.2.5 What’s new in 1.3.1 (July 25, 2021)

These are the changes in pandas 1.3.1. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Pandas could not be built on PyPy (GH42355)
- DataFrame constructed with an older version of pandas could not be unpickled (GH42345)
- Performance regression in constructing a DataFrame from a dictionary of dictionaries (GH42248)
- Fixed regression in DataFrame.agg() dropping values when the DataFrame had an Extension Array dtype, a duplicate index, and axis=1 (GH42380)
- Fixed regression in DataFrame.astype() changing the order of noncontiguous data (GH42396)
- Performance regression in DataFrame in reduction operations requiring casting such as DataFrame.mean() on integer data (GH38592)
- Performance regression in DataFrame.to_dict() and Series.to_dict() when orient argument one of “records”, “dict”, or “split” (GH42352)
- Fixed regression in indexing with a list subclass incorrectly raising TypeError (GH42433, GH42461)
- Fixed regression in DataFrame.isin() and Series.isin() raising TypeError with nullable data containing at least one missing value (GH42405)
- Regression in concat() between objects with bool dtype and integer dtype casting to object instead of to integer (GH42092)
- Bug in Series constructor not accepting a dask.Array (GH38645)
- Fixed regression for SettingWithCopyWarning displaying incorrect stacklevel (GH42570)
- Fixed regression for merge_asof() raising KeyError when one of the by columns is in the index (GH34488)
- Fixed regression in to_datetime() returning pd.NaT for inputs that produce duplicated values, when cache=True (GH42259)
- Fixed regression in SeriesGroupBy.value_counts() that resulted in an IndexError when called on a Series with one row (GH42618)

Bug fixes

- Fixed bug in DataFrame.transpose() dropping values when the DataFrame had an Extension Array dtype and a duplicate index (GH42380)
- Fixed bug in DataFrame.to_xml() raising KeyError when called with index=False and an offset index (GH42458)
- Fixed bug in Styler.set_sticky() not handling index names correctly for single index columns case (GH42537)
- Fixed bug in DataFrame.copy() failing to consolidate blocks in the result (GH42579)
Contributors

A total of 17 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Fangchen Li
- GYvan +
- Matthew Roeschke
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler
- Richard Shadrach
- Shoham Debnath +
- Simon Hawkins
- Stephan Heßelmann +
- Stephen +
- Thomas Li
- Zheyuan +
- attack68
- jbrockmendel
- neelmraman +

5.2.6 What’s new in 1.3.0 (July 2, 2021)

These are the changes in pandas 1.3.0. See Release notes for a full changelog including other versions of pandas.

**Warning:** When reading new Excel 2007+ (.xlsx) files, the default argument `engine=None to read_excel()` will now result in using the `openpyxl` engine in all cases when the option `io.excel.xlsx.reader` is set to "auto". Previously, some cases would use the `xlrd` engine instead. See What’s new 1.2.0 for background on this change.

Enhancements

Custom HTTP(s) headers when reading csv or json files

When reading from a remote URL that is not handled by `fsspec` (e.g. HTTP and HTTPS) the dictionary passed to `storage_options` will be used to create the headers included in the request. This can be used to control the User-Agent header or send other custom headers (GH36688). For example:

```
In [1]: headers = {'User-Agent': "pandas"}
In [2]: df = pd.read_csv(
```
Read and write XML documents

We added I/O support to read and render shallow versions of XML documents with `read_xml()` and `DataFrame.to_xml()`. Using lxml as parser, both XPath 1.0 and XSLT 1.0 are available. (GH27554)

```
In [1]: xml = """<?xml version='1.0' encoding='utf-8'?>
...:   <data>
...:     <row>
...:       <shape>square</shape>
...:       <degrees>360</degrees>
...:       <sides>4.0</sides>
...:     </row>
...:     <row>
...:       <shape>circle</shape>
...:       <degrees>360</degrees>
...:       <sides/>
...:     </row>
...:     <row>
...:       <shape>triangle</shape>
...:       <degrees>180</degrees>
...:       <sides>3.0</sides>
...:     </row>
...: </data>"""
In [2]: df = pd.read_xml(xml)
In [3]: df
Out[3]:
         shape  degrees  sides
0   square     360    4.0
1   circle     360       NaN
2  triangle    180    3.0
In [4]: df.to_xml()
Out[4]:
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```
For more, see *Writing XML* in the user guide on IO tools.

**Styler enhancements**

We provided some focused development on *Styler*. See also the *Styler documentation* which has been revised and improved (GH39720, GH39317, GH40493).

- The method *Styler.set_table_styles()* can now accept more natural CSS language for arguments, such as 'color:red;' instead of ['color', 'red'] (GH39563)
- The methods *Styler.highlight_null(), Styler.highlight_min(), and Styler.highlight_max()* now allow custom CSS highlighting instead of the default background coloring (GH40242)
- *Styler.apply()* now accepts functions that return an ndarray when axis=None, making it now consistent with the axis=0 and axis=1 behavior (GH39359)
- When incorrectly formatted CSS is given via *Styler.apply()* or *Styler.applymap()* an error is now raised upon rendering (GH39660)
- *Styler.format()* now accepts the keyword argument *escape* for optional HTML and LaTeX escaping (GH40388, GH41619)
- *Styler.background_gradient()* has gained the argument *gmap* to supply a specific gradient map for shading (GH22727)
- *Styler.clear()* now clears *Styler.hidden_index* and *Styler.hidden_columns* as well (GH40484)
- Added the method *Styler.highlight_between()* (GH39821)
- Added the method *Styler.highlight_quantile()* (GH40926)
- Added the method *Styler.text_gradient()* (GH41098)
- Added the method *Styler.set_tooltips()* to allow hover tooltips; this can be used enhance interactive displays (GH21266, GH40284)
- Added the parameter *precision* to the method *Styler.format()* to control the display of floating point numbers (GH40134)
- *Styler* rendered HTML output now follows the w3 HTML Style Guide (GH39626)
- Many features of the *Styler* class are now either partially or fully usable on a DataFrame with a non-unique indexes or columns (GH41143)
- One has greater control of the display through separate sparsification of the index or columns using the *new styler options*, which are also usable via *option_context()* (GH41142)
- Added the option *styler.render.max_elements* to avoid browser overload when styling large DataFrames (GH40712)
• Added the method `Styler.to_latex()` (GH21673, GH42320), which also allows some limited CSS conversion (GH40731)
• Added the method `Styler.to_html()` (GH13379)
• Added the method `Styler.set_sticky()` to make index and column headers permanently visible in scrolling HTML frames (GH29072)

**DataFrame constructor honors `copy=False` with dict**

When passing a dictionary to `DataFrame` with `copy=False`, a copy will no longer be made (GH32960).

```
In [3]: arr = np.array([1, 2, 3])
In [4]: df = pd.DataFrame({"A": arr, "B": arr.copy()}, copy=False)

In [5]: df
Out[5]:
       A   B
0     1   1
1     2   2
2     3   3
```

df["A"] remains a view on arr:

```
In [6]: arr[0] = 0
In [7]: assert df.iloc[0, 0] == 0
```

The default behavior when not passing `copy` will remain unchanged, i.e. a copy will be made.

**PyArrow backed string data type**

We've enhanced the `StringDtype`, an extension type dedicated to string data. (GH39908)

It is now possible to specify a `storage` keyword option to `StringDtype`. Use pandas options or specify the dtype using `dtype='string[pyarrow]'` to allow the StringArray to be backed by a PyArrow array instead of a NumPy array of Python objects.

The PyArrow backed StringArray requires pyarrow 1.0.0 or greater to be installed.

```
Warning: string[pyarrow] is currently considered experimental. The implementation and parts of the API may change without warning.
```

```
In [8]: pd.Series(["abc", None, 'def'], dtype=pd.StringDtype(storage="pyarrow"))
Out[8]:
0   abc
1  <NA>
2   def
dtype: string
```

You can use the alias "string[pyarrow]" as well.
In [9]: s = pd.Series(['abc', None, 'def'], dtype="string[pyarrow"]

In [10]: s
Out[10]:
0    abc
1    <NA>
2    def
dtype: string

You can also create a PyArrow backed string array using pandas options.

In [11]: with pd.option_context("string_storage", "pyarrow"):
In [11]: s = pd.Series(['abc', None, 'def'], dtype="string")

In [12]: s
Out[12]:
0    abc
1    <NA>
2    def
dtype: string

The usual string accessor methods work. Where appropriate, the return type of the Series or columns of a DataFrame will also have string dtype.

In [13]: s.str.upper()
Out[13]:
0    ABC
1    <NA>
2    DEF
dtype: string

In [14]: s.str.split('b', expand=True).dtypes
Out[14]:
0    string
1    string
dtype: object

String accessor methods returning integers will return a value with Int64Dtype

In [15]: s.str.count("a")
Out[15]:
0    1
1    <NA>
2    0
dtype: Int64
Centered datetime-like rolling windows

When performing rolling calculations on DataFrame and Series objects with a datetime-like index, a centered datetime-like window can now be used (GH38780). For example:

```python
In [16]: df = pd.DataFrame(
   ....:     {"A": [0, 1, 2, 3, 4]}, index=pd.date_range("2020", periods=5, freq="1D")
   ....: )
   ....:
In [17]: df
Out[17]:
    A
2020-01-01  0
2020-01-02  1
2020-01-03  2
2020-01-04  3
2020-01-05  4
In [18]: df.rolling("2D", center=True).mean()
Out[18]:
    A
2020-01-01  0.5
2020-01-02  1.5
2020-01-03  2.5
2020-01-04  3.5
2020-01-05  4.0
```

Other enhancements

- `DataFrame.rolling()`, `Series.rolling()`, `DataFrame.expanding()`, and `Series.expanding()` now support a method argument with a 'table' option that performs the windowing operation over an entire DataFrame. See Window Overview for performance and functional benefits (GH15095, GH38995)
- ExponentialMovingWindow now support a online method that can perform mean calculations in an online fashion. See Window Overview (GH41673)
- Added `MultiIndex.dtypes()` (GH37062)
- Added end and end_day options for the origin argument in `DataFrame.resample()` (GH37804)
- Improved error message when usecols and names do not match for `read_csv()` and engine="c" (GH29042)
- Improved consistency of error messages when passing an invalid win_type argument in Window methods (GH15969)
- `read_sql_query()` now accepts a dtype argument to cast the columnar data from the SQL database based on user input (GH10285)
- `read_csv()` now raising ParserWarning if length of header or given names does not match length of data when usecols is not specified (GH21768)
- Improved integer type mapping from pandas to SQLAlchemy when using `DataFrame.to_sql()` (GH35076)
- `to_numeric()` now supports downcasting of nullable ExtensionDtype objects (GH33013)
- Added support for dict-like names in `MultiIndex.set_names and MultiIndex.rename` (GH20421)
*read_excel()* can now auto-detect .xlsxb files and older .xls files (GH35416, GH41225)

*ExcelWriter* now accepts an *if_sheet_exists* parameter to control the behavior of append mode when writing to existing sheets (GH40230)

*Rolling.sum()*.  *Expanding.sum()*.* Rolling.mean()*.* Expanding.mean()*.* ExponentialMovingWindow.mean()*.* Rolling.median()*.* Expanding.median()*.* Rolling.max()*.* Expanding.max()*.* Rolling.min()* and *Expanding.min()* now support Numba execution with the *engine* keyword (GH38895, GH41267)

*DataFrame.apply()* can now accept NumPy unary operators as strings, e.g. *df.apply("sqrt")*, which was already the case for *Series.apply()* (GH39116)

*DataFrame.apply()* can now accept non-callable DataFrame properties as strings, e.g. *df.apply("size")*, which was already the case for *Series.apply()* (GH39116)

*DataFrame.applymap()* can now accept kwargs to pass on to the user-provided *func* (GH39987)

Passing a *DataFrame* indexer to *iloc* is now disallowed for *Series.__getitem__()* and *DataFrame.__getitem__()* (GH39004)

*Series.apply()* can now accept list-like or dictionary-like arguments that aren’t lists or dictionaries, e.g. *ser.apply(np.array(["sum", "mean"]))*, which was already the case for *DataFrame.apply()* (GH39140)

*DataFrame.plot.scatter()* can now accept a categorical column for the argument *c* (GH12380, GH31357)

*Series.loc()* now raises a helpful error message when the Series has a *MultiIndex* and the indexer has too many dimensions (GH35349)

*read_stata()* now supports reading data from compressed files (GH26599)

Added support for parsing ISO 8601-like timestamps with negative signs to *Timedelta* (GH37172)

Added support for unary operators in *FloatingArray* (GH38749)

*RangeIndex* can now be constructed by passing a *range* object directly e.g. *pd.RangeIndex(range(3))* (GH12067)

*Series.round()* and *DataFrame.round()* now work with nullable integer and floating dtypes (GH38844)

*read_csv()* and *read_json()* expose the argument *encoding_errors* to control how encoding errors are handled (GH39450)

*GroupBy.any()* and *GroupBy.all()* use Kleene logic with nullable data types (GH37506)

*GroupBy.any()* and *GroupBy.all()* return a *BooleanDtype* for columns with nullable data types (GH33449)

*GroupBy.any()* and *GroupBy.all()* raising with *object* data containing pd.NA even when skipna=True (GH37501)

*GroupBy.rank()* now supports object-dtype data (GH38278)

Constructing a *DataFrame* or *Series* with the *data* argument being a Python iterable that is *not* a NumPy *ndarray* consisting of NumPy scalars will now result in a data type with a precision the maximum of the NumPy scalars; this was already the case when *data* is a NumPy *ndarray* (GH40908)

Add keyword *sort* to *pivot_table()* to allow non-sorting of the result (GH39143)

Add keyword *dropna* to *DataFrame.value_counts()* to allow counting rows that include NA values (GH41325)

*Series.replace()* will now cast results to *PeriodDtype* where possible instead of *object* dtype (GH41526)

Improved error message in *corr* and *cov* methods on *Rolling*, *Expanding*, and *ExponentialMovingWindow* when *other* is not a *DataFrame* or *Series* (GH41741)
Series.between() can now accept left or right as arguments to inclusive to include only the left or right boundary (GH40245)

DataFrame.explode() now supports exploding multiple columns. Its column argument now also accepts a list of str or tuples for exploding on multiple columns at the same time (GH39240)

DataFrame.sample() now accepts the ignore_index argument to reset the index after sampling, similar to DataFrame.drop_duplicates() and DataFrame.sort_values() (GH38581)

Notable bug fixes

These are bug fixes that might have notable behavior changes.

Categorical.unique now always maintains same dtype as original

Previously, when calling Categorical.unique() with categorical data, unused categories in the new array would be removed, making the dtype of the new array different than the original (GH18291)

As an example of this, given:

```
In [19]: dtype = pd.CategoricalDtype(['bad', 'neutral', 'good'], ordered=True)
In [20]: cat = pd.Categorical(['good', 'good', 'bad', 'bad'], dtype=dtype)
In [21]: original = pd.Series(cat)
In [22]: unique = original.unique()
```

**Previous behavior:**

```
In [1]: unique
['good', 'bad']
Categories (2, object): ['bad' < 'good']
In [2]: original.dtype == unique.dtype
False
```

**New behavior:**

```
In [23]: unique
['good', 'bad']
Categories (3, object): ['bad' < 'neutral' < 'good']
In [24]: original.dtype == unique.dtype
Out[24]: True
```
Preserve dtypes in DataFrame.combine_first()

*DataFrame.combine_first()* will now preserve dtypes (GH7509)

```python
In [25]: df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 2, 3]}, index=[0, 1, 2])
```

```text
In [26]: df1
Out[26]:
   A  B
0  1  1
1  2  2
2  3  3
```

```python
In [27]: df2 = pd.DataFrame({'B': [4, 5, 6], 'C': [1, 2, 3]}, index=[2, 3, 4])
```

```text
In [28]: df2
Out[28]:
   B  C
2  4  1
3  5  2
4  6  3
```

```python
In [29]: combined = df1.combine_first(df2)
```

**Previous behavior:**

```python
In [1]: combined.dtypes
Out[2]:
A  float64
B  float64
C  float64
dtype: object
```

**New behavior:**

```python
In [30]: combined.dtypes
Out[30]:
A  float64
B  int64
C  float64
dtype: object
```

**Groupby methods agg and transform no longer changes return dtype for callables**

Previously the methods *DataFrameGroupBy.aggregate()*, *SeriesGroupBy.aggregate()*, *DataFrameGroupBy.transform()*, and *SeriesGroupBy.transform()* might cast the result dtype when the argument *func* is callable, possibly leading to undesirable results (GH21240). The cast would occur if the result is numeric and casting back to the input dtype does not change any values as measured by *np.allclose*. Now no such casting occurs.

```python
In [31]: df = pd.DataFrame({'key': [1, 1], 'a': [True, False], 'b': [True, True]})
In [32]: df
```
`pandas`: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

```
Out[32]:
   key   a   b
0   1  True  True
1   1 False  True
```

**Previous behavior:**

```
In [5]: df.groupby('key').agg(lambda x: x.sum())
Out[5]:
   a   b
key
  1  True  2
```

**New behavior:**

```
In [33]: df.groupby('key').agg(lambda x: x.sum())
Out[33]:
   a   b
key
  1   1  2
```

**Float result for `GroupBy.mean()`, `GroupBy.median()`, and `GroupBy.var()`**

Previously, these methods could result in different dtypes depending on the input values. Now, these methods will always return a float dtype. (GH41137)

```
In [34]: df = pd.DataFrame({'a': [True], 'b': [1], 'c': [1.0]})

Previous behavior:

```
In [5]: df.groupby(df.index).mean()
Out[5]:
   a   b   c
0  True  1  1.0
```

New behavior:

```
In [35]: df.groupby(df.index).mean()
Out[35]:
   a   b   c
0  1.0  1.0  1.0
```
Try operating inplace when setting values with `loc` and `iloc`

When setting an entire column using `loc` or `iloc`, pandas will try to insert the values into the existing data rather than create an entirely new array.

```python
In [36]: df = pd.DataFrame(range(3), columns=["A"], dtype="float64")
In [37]: values = df.values
In [38]: new = np.array([5, 6, 7], dtype="int64")
In [39]: df.loc[[0, 1, 2], "A"] = new
```

In both the new and old behavior, the data in `values` is overwritten, but in the old behavior the dtype of `df["A"]` changed to `int64`.

*Previous behavior:*

```python
In [1]: df.dtypes
Out[1]:
A  int64
dtype: object
In [2]: np.shares_memory(df["A"].values, new)
Out[2]: False
In [3]: np.shares_memory(df["A"].values, values)
Out[3]: False
```

In pandas 1.3.0, `df` continues to share data with `values`

*New behavior:*

```python
In [40]: df.dtypes
Out[40]:
A  float64
dtype: object
In [41]: np.shares_memory(df["A"], new)
Out[41]: False
In [42]: np.shares_memory(df["A"], values)
Out[42]: True
```

Never operate inplace when setting `frame[keys] = values`

When setting multiple columns using `frame[keys] = values` new arrays will replace pre-existing arrays for these keys, which will not be over-written (GH39510). As a result, the columns will retain the dtype(s) of `values`, never casting to the dtypes of the existing arrays.

```python
In [43]: df = pd.DataFrame(range(3), columns=["A"], dtype="float64")
In [44]: df["A"] = 5
```

In the old behavior, 5 was cast to `float64` and inserted into the existing array backing `df`:

*Previous behavior:*
In [1]: df.dtypes
Out[1]:
A  float64

In the new behavior, we get a new array, and retain an integer-dtyped 5:

New behavior:

In [45]: df.dtypes
Out[45]:
A  int64
dtype: object

Consistent casting with setting into Boolean Series

Setting non-boolean values into a Series with dtype=bool now consistently casts to dtype=object (GH38709)

In [46]: orig = pd.Series([True, False])
In [47]: ser = orig.copy()
In [48]: ser.iloc[1] = np.nan
In [49]: ser2 = orig.copy()
In [50]: ser2.iloc[1] = 2.0

Previous behavior:

In [1]: ser
Out [1]:
0  1.0
1  NaN
dtype: float64

In [2]: ser2
Out [2]:
0  True
1  2.0
dtype: object

New behavior:

In [51]: ser
Out[51]:
0  True
1  NaN
dtype: object

In [52]: ser2
Out[52]:
0  True
(continues on next page)
GroupBy.rolling no longer returns grouped-by column in values

The group-by column will now be dropped from the result of a `groupby.rolling` operation (GH32262)

\[
\text{In [53]: df = pd.DataFrame({"A": [1, 1, 2, 3], "B": [0, 1, 2, 3]})}
\]

\[
\text{In [54]: df}
\]

\[
\text{Out[54]:}
\]

\[
\begin{array}{ll}
A & B \\
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
\end{array}
\]

Previous behavior:

\[
\text{In [1]: df.groupby("A").rolling(2).sum()}
\]

\[
\text{Out[1]:}
\]

\[
\begin{array}{llll}
A & B \\
1 & NaN & NaN \\
1 & 2.0 & 1.0 \\
2 & NaN & NaN \\
3 & NaN & NaN \\
\end{array}
\]

New behavior:

\[
\text{In [55]: df.groupby("A").rolling(2).sum()}
\]

\[
\text{Out[55]:}
\]

\[
\begin{array}{lll}
B \\
A \\
1 & NaN \\
1 & 1.0 \\
2 & NaN \\
3 & NaN \\
\end{array}
\]

Removed artificial truncation in rolling variance and standard deviation

`Rolling.std()` and `Rolling.var()` will no longer artificially truncate results that are less than \(~1\times10^{-8}\) and \(~1\times10^{-15}\) respectively to zero (GH37051, GH40448, GH39872).

However, floating point artifacts may now exist in the results when rolling over larger values.

\[
\text{In [56]: s = pd.Series([7, 5, 5, 5])}
\]

\[
\text{In [57]: s.rolling(3).var()}
\]

\[
\text{Out[57]:}
\]

(continues on next page)
GroupBy.rolling with MultiIndex no longer drops levels in the result

GroupBy.rolling() will no longer drop levels of a DataFrame with a MultiIndex in the result. This can lead to a perceived duplication of levels in the resulting MultiIndex, but this change restores the behavior that was present in version 1.1.3 (GH38787, GH38523).

```python
In [58]: index = pd.MultiIndex.from_tuples([(idx1, idx2)], names=[label1, label2])
In [59]: df = pd.DataFrame({'a': [1], 'b': [2]}, index=index)
In [60]: df
Out[60]:
    a  b
label1 label2
idx1 idx2  1  2

Previous behavior:

```python
In [1]: df.groupby('label1').rolling(1).sum()
Out[1]:
     a  b
label1
idx1  1.0  2.0
```

New behavior:

```python
In [61]: df.groupby('label1').rolling(1).sum()
Out[61]:
     a  b
label1 label1 label2
idx1 idx1 idx2  1.0  2.0
```

Backwards incompatible API changes

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated. If installed, we now require:
For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.

```plaintext
<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.17.3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pytz</td>
<td>2017.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.7.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.7.0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>6.0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mypy (dev)</td>
<td>0.812</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>setuptools</td>
<td>38.6.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

See Dependencies and Optional dependencies for more.

**Other API changes**

- Partially initialized `CategoricalDtype` objects (i.e. those with `categories=None`) will no longer compare as equal to fully initialized dtype objects (GH38516)
- Accessing `_constructor_expanddim` on a `DataFrame` and `_constructor_sliced` on a `Series` now raise an `AttributeError`. Previously a `NotImplementedError` was raised (GH38782)
- Added new engine and **`engine_kwargs`** parameters to `DataFrame.to_sql()` to support other future “SQL engines”. Currently we still only use SQLAlchemy under the hood, but more engines are planned to be supported such as turbodbc (GH36893)
- Removed redundant `freq` from `PeriodIndex` string representation (GH41653)
pandas: powerful Python data analysis toolkit, Release 1.4.2

• ExtensionDtype.construct_array_type() is now a required method instead of an optional one for ExtensionDtype subclasses (GH24860)

• Calling hash on non-hashable pandas objects will now raise TypeError with the built-in error message (e.g. unhashable type: 'Series'). Previously it would raise a custom message such as 'Series' objects are mutable, thus they cannot be hashed. Furthermore, isinstance(<Series>, abc.collections.Hashable) will now return False (GH40013)

• Styler.from_custom_template() now has two new arguments for template names, and removed the old name, due to template inheritance having been introducing for better parsing (GH42053). Subclassing modifications to Styler attributes are also needed.

Build

• Documentation in .pptx and .pdf formats are no longer included in wheels or source distributions. (GH30741)

Deprecations

Deprecated dropping nuisance columns in DataFrame reductions and DataFrameGroupBy operations

Calling a reduction (e.g. .min, .max, .sum) on a DataFrame with numeric_only=None (the default), columns where the reduction raises a TypeError are silently ignored and dropped from the result.

This behavior is deprecated. In a future version, the TypeError will be raised, and users will need to select only valid columns before calling the function.

For example:

```python
In [62]: df = pd.DataFrame({"A": [1, 2, 3, 4], "B": pd.date_range("2016-01-01", periods=4)})

In [63]: df
Out[63]:
   A    B
0  1 2016-01-01
1  2 2016-01-02
2  3 2016-01-03
3  4 2016-01-04

Old behavior:

In [3]: df.prod()
Out[3]:
   A
0 24
dtype: int64

Future behavior:

In [4]: df.prod()
   ...  # continues on next page
TypeError: 'DatetimeArray' does not implement reduction 'prod'  
(continues on next page)
Similarly, when applying a function to `DataFrameGroupBy`, columns on which the function raises `TypeError` are currently silently ignored and dropped from the result.

This behavior is deprecated. In a future version, the `TypeError` will be raised, and users will need to select only valid columns before calling the function.

For example:

```python
In [64]: df = pd.DataFrame({"A": [1, 2, 3, 4], "B": pd.date_range("2016-01-01", periods=4)})
In [65]: gb = df.groupby([1, 1, 2, 2])
```

**Old behavior:**

```python
In [4]: gb.prod(numeric_only=False)
Out[4]:
A
  1  2
  2 12
```

**Future behavior:**

```python
In [5]: gb.prod(numeric_only=False)
...  # TypeError: datetime64 type does not support prod operations
In [6]: gb["A"].prod(numeric_only=False)
Out[6]:
A
  1  2
  2 12
```

### Other Deprecations

- Deprecated allowing scalars to be passed to the `Categorical` constructor (GH38433)
- Deprecated constructing `CategoricalIndex` without passing list-like data (GH38944)
- Deprecated allowing subclass-specific keyword arguments in the `Index` constructor, use the specific subclass directly instead (GH14093, GH21311, GH22315, GH26974)
- Deprecated the `astype()` method of datetimelike (`timedelta64[ns]`, `datetime64[ns]`, `Datetime64TZDtype`, `PeriodDtype`) to convert to integer dtypes, use `values.view(...)` instead (GH38544). This deprecation was later reverted in pandas 1.4.0.
- Deprecated `MultiIndex.is_lexsorted()` and `MultiIndex.lexsort_depth()`, use `MultiIndex.is_monotonic_increasing()` instead (GH32259)
• Deprecated keyword try_cast in Series.where(), Series.mask(), DataFrame.where(), DataFrame.mask(): cast results manually if desired (GH38836)

• Deprecated comparison of Timestamp objects with datetime.date objects. Instead of e.g. ts <= mydate use ts <= pd.Timestamp(mydate) or ts.date() <= mydate (GH36131)

• Deprecated Rolling.win_type returning "freq" (GH38963)

• Deprecated DataFrame indexer for Series.__setitem__() and DataFrame.__setitem__() (GH39004)

• Deprecated ExponentialMovingWindow.vol() (GH39220)

• Using .astype to convert between datetime64[ns] dtype and DatetimeTZDtype is deprecated and will raise in a future version, use obj.tz_localize or obj.dt.tz_localize instead (GH38622)

• Deprecated casting datetime.date objects to datetime64 when used as fill_value in DataFrame.unstack(), DataFrame.shift(), Series.shift(), and DataFrame.reindex(), pass pd.Timestamp(dateobj) instead (GH39767)

• Deprecated Styler.set_na_rep() and Styler.set_precision() in favor of Styler.format() with na_rep and precision as existing and new input arguments respectively (GH40134, GH40425)

• Deprecated allowing partial failure in Series.transform() and DataFrame.transform() when func is list-like or dict-like and raises anything but TypeError: func raising anything but a TypeError will raise in a future version (GH40211)

• Deprecated arguments error_bad_lines and warn_bad_lines in read_csv() and read_table() in favor of argument on_bad_lines (GH15122)

• Deprecated support for np.ma.mrecords.MaskedRecords in the DataFrame constructor, pass {name: data[name] for name in data.dtype.names} instead (GH40363)

• Deprecated using merge(), DataFrame.merge(), and DataFrame.join() on a different number of levels (GH34862)

• Deprecated the use of **kwargs in ExcelWriter; use the keyword argument engine_kwars instead (GH40430)

• Deprecated the level keyword for DataFrame and Series aggregations; use groupby instead (GH39983)

• Deprecated the inplace parameter of Categorical.remove_categories(), Categorical.add_categories(), Categorical.reorder_categories(), Categorical.rename_categories(), and Categorical.set_categories() and will be removed in a future version (GH37643)

• Deprecated merge() producing duplicated columns through the suffixes keyword and already existing columns (GH22818)

• Deprecated setting Categorical._codes, create a new Categorical with the desired codes instead (GH40606)

• Deprecated the convert_float optional argument in read_excel() and ExcelFile.parse() (GH41127)

• Deprecated behavior of DatetimeIndex.union() with mixed timezones; in a future version both will be cast to UTC instead of object dtype (GH39328)

• Deprecated using usecols with out of bounds indices for read_csv() with engine="c" (GH25623)

• Deprecated special treatment of lists with first element a Categorical in the DataFrame constructor; pass as pd.DataFrame({col: categorical, ...}) instead (GH38845)
• Deprecated behavior of `DataFrame` constructor when a `dtype` is passed and the data cannot be cast to that `dtype`. In a future version, this will raise instead of being silently ignored (GH24435)

• Deprecated the `Timestamp.freq` attribute. For the properties that use it (`is_month_start`, `is_month_end`, `is_quarter_start`, `is_quarter_end`, `is_year_start`, `is_year_end`), when you have a `freq` use e.g. `freq.is_month_start(ts)` (GH15146)

• Deprecated construction of `Series` or `DataFrame` with `DatetimeTZDtype` data and `datetime64[ns]` dtype. Use `Series(data).dt.tz_localize(None)` instead (GH41555, GH33401)

• Deprecated behavior of `Series` construction with large integer values and small integer dtype silently overflowing; use `Series(data).astype(dtype)` instead (GH41734)

• Deprecated inference of `timedelta64[ns]`, `datetime64[ns]`, or `DatetimeTZDtype` dtypes in `Series` construction when data containing strings is passed and no `dtype` is passed (GH33558)

• In a future version, constructing `Series` or `DataFrame` with `datetime64[ns]` data and `DatetimeTZDtype` will treat the data as wall-times instead of as UTC times (matching DatetimeIndex behavior). To treat the data as UTC times, use `pd.Series(data).dt.tz_localize("UTC").dt.tz_convert(dtype.tz)` or `pd.Series(data.view("int64"), dtype=dtype)` (GH33401)

• Deprecated passing lists as key to `DataFrame.xs()` and `Series.xs()` (GH41760)

• Deprecated boolean arguments of `inclusive` in `Series.between()` to have `{"left", "right", "neither", "both"}` as standard argument values (GH40628)

• Deprecated passing arguments as positional for all of the following, with exceptions noted (GH41485):
  - `concat()` (other than `objs`)
  - `read_csv()` (other than `filepath_or_buffer`)
  - `read_table()` (other than `filepath_or_buffer`)
  - `DataFrame.clip()` and `Series.clip()` (other than `upper` and `lower`)
  - `DataFrame.drop_duplicates()` (except for `subset`), `Series.drop_duplicates()`, `Index.drop_duplicates()` and `MultiIndex.drop_duplicates()`
  - `DataFrame.drop()` (other than `labels`) and `Series.drop()`
  - `DataFrame.dropna()` and `Series.dropna()`
  - `DataFrame.ffill()`, `Series.ffill()`, `DataFrame.bfill()`, and `Series.bfill()`
  - `DataFrame.fillna()` and `Series.fillna()` (apart from `value`)
  - `DataFrame.interpolate()` and `Series.interpolate()` (other than `method`)
  - `DataFrame.mask()` and `Series.mask()` (other than `cond` and `other`)
  - `DataFrame.reset_index()` (other than `level`) and `Series.reset_index()`
  - `DataFrame.set_axis()` and `Series.set_axis()` (other than `labels`)
  - `DataFrame.set_index()` (other than `keys`)
  - `DataFrame.sort_index()` and `Series.sort_index()`
  - `DataFrame.sort_values()` (other than `by`) and `Series.sort_values()`
  - `DataFrame.where()` and `Series.where()` (other than `cond` and `other`)
  - `Index.set_names()` and `MultiIndex.set_names()` (except for `names`)

5.2. Version 1.3
pandas: powerful Python data analysis toolkit, Release 1.4.2

- `MultiIndex.codes()` (except for `codes`)
- `MultiIndex.set_levels()` (except for `levels`)
- `Resampler.interpolate()` (other than `method`)

Performance improvements

- Performance improvement in `IntervalIndex.isin()` (GH38353)
- Performance improvement in `Series.mean()` for nullable data types (GH34814)
- Performance improvement in `Series.isin()` for nullable data types (GH38340)
- Performance improvement in `DataFrame.fillna()` with `method="pad"` or `method="backfill"` for nullable floating and nullable integer dtypes (GH39953)
- Performance improvement in `DataFrame.corr()` for `method=kendall` (GH28329)
- Performance improvement in `DataFrame.corr()` for `method=spearman` (GH40956, GH41885)
- Performance improvement in `Rolling.corr()` and `Rolling.cov()` (GH39388)
- Performance improvement in `RollingGroupby.corr()`, `ExpandingGroupby.corr()`, `ExpandingGroupby.corr()` and `ExpandingGroupby.cov()` (GH39591)
- Performance improvement in `unique()` for object data type (GH37615)
- Performance improvement in `json_normalize()` for basic cases (including separators) (GH40035 GH15621)
- Performance improvement in `ExpandingGroupby` aggregation methods (GH39664)
- Performance improvement in `Styler` where render times are more than 50% reduced and now matches `DataFrame.to_html()` (GH39972 GH39952, GH40425)
- The method `Styler.set_td_classes()` is now as performant as `Styler.apply()` and `Styler.applymap()`, and even more so in some cases (GH40453)
- Performance improvement in `ExponentialMovingWindow.mean()` with `times` (GH39784)
- Performance improvement in `GroupBy.apply()` when requiring the Python fallback implementation (GH40176)
- Performance improvement in the conversion of a PyArrow Boolean array to a pandas nullable Boolean array (GH40151)
- Performance improvement for concatenation of data with type `CategoricalDtype` (GH40193)
- Performance improvement in `GroupBy.cummin()` and `GroupBy.cummax()` with nullable data types (GH37493)
- Performance improvement in `Series.nunique()` with nan values (GH40865)
- Performance improvement in `DataFrame.transpose()`, `Series.unstack()` with `DatetimeTZDtype` (GH40149)
- Performance improvement in `Series.plot()` and `DataFrame.plot()` with entry point lazy loading (GH41492)
Bug fixes

Categorical

- Bug in `CategoricalIndex` incorrectly failing to raise `TypeError` when scalar data is passed (GH38614)
- Bug in `CategoricalIndex.reindex` failed when the `Index` passed was not categorical but whose values were all labels in the category (GH28690)
- Bug where constructing a `Categorical` from an object-dtype array of `date` objects did not round-trip correctly with `astype` (GH38552)
- Bug in constructing a `DataFrame` from an `ndarray` and a `CategoricalDtype` (GH38857)
- Bug in setting categorical values into an object-dtype column in a `DataFrame` (GH39136)
- Bug in `DataFrame.reindex()` was raising an `IndexError` when the new index contained duplicates and the old index was a `CategoricalIndex` (GH38906)
- Bug in `Categorical.fillna()` with a tuple-like category raising `NotImplementedError` instead of `ValueError` when filling with a non-category tuple (GH41914)

Datetimelike

- Bug in `DataFrame` and `Series` constructors sometimes dropping nanoseconds from `Timestamp` (resp. `Timedelta`) data, with `dtype=datet ime64[ns]` (resp. `timedelta64[ns]`) (GH38032)
- Bug in `DataFrame.first()` and `Series.first()` with an offset of one month returning an incorrect result when the first day is the last day of a month (GH29623)
- Bug in constructing a `DataFrame` or `Series` with mismatched `datetime64` data and `timedelta64` `dtype`, or vice-versa, failing to raise a `TypeError` (GH3875, GH38764, GH38792)
- Bug in constructing a `Series` or `DataFrame` with a `datetime` object out of bounds for `datetime64[ns]` `dtype` or a `timedelta` object out of bounds for `timedelta64[ns]` `dtype` (GH38792, GH38965)
- Bug in `DatetimeIndex.intersection()`, `DatetimeIndex.symmetric_difference()`, `PeriodIndex.intersection()`, `PeriodIndex.symmetric_difference()` always returning object-dtype when operating with `CategoricalIndex` (GH38741)
- Bug in `DatetimeIndex.intersection()` giving incorrect results with non-Tick frequencies with n != 1 (GH42104)
- Bug in `Series.where()` incorrectly casting `datetime64` values to `int64` (GH37682)
- Bug in `Categorical` incorrectly typecasting `datetime` object to `Timestamp` (GH38878)
- Bug in comparisons between `Timestamp` object and `datetime64` objects just outside the implementation bounds for nanosecond `datetime64` (GH39221)
- Bug in `Timestamp.round()`, `Timestamp.floor()`, `Timestamp.ceil()` for values near the implementation bounds of `Timestamp` (GH39244)
- Bug in `Timedelta.round()`, `Timedelta.floor()`, `Timedelta.ceil()` for values near the implementation bounds of `Timedelta` (GH38964)
- Bug in `date_range()` incorrectly creating `DatetimeIndex` containing `NaT` instead of raising `OutOfBoundsDatetime` in corner cases (GH24124)
- Bug in `infer_freq()` incorrectly fails to infer ‘H’ frequency of `DatetimeIndex` if the latter has a timezone and crosses DST boundaries (GH39556)

5.2. Version 1.3
- Bug in `Series` backed by `DatetimeArray` or `TimedeltaArray` sometimes failing to set the array's `freq` to `None` (GH41425)

**Timedelta**

- Bug in constructing `Timedelta` from `np.timedelta64` objects with non-nanosecond units that are out of bounds for `timedelta64[ns]` (GH38965)
- Bug in constructing a `TimedeltaIndex` incorrectly accepting `np.datetime64("NaT")` objects (GH39462)
- Bug in constructing `Timedelta` from an input string with only symbols and no digits failed to raise an error (GH39710)
- Bug in `TimedeltaIndex` and `to_timedelta()` failing to raise when passed non-nanosecond `timedelta64` arrays that overflow when converting to `timedelta64[ns]` (GH40008)

**Timezones**

- Bug in different `tzinfo` objects representing UTC not being treated as equivalent (GH39216)
- Bug in `dateutil.tz.gettz("UTC")` not being recognized as equivalent to other UTC-representing `tzinfo`s (GH39276)

**Numeric**

- Bug in `DataFrame.quantile()`, `DataFrame.sort_values()` causing incorrect subsequent indexing behavior (GH38351)
- Bug in `DataFrame.sort_values()` raising an `IndexError` for empty by (GH40258)
- Bug in `DataFrame.select_dtypes()` with `include=np.number` would drop numeric `ExtensionDtype` columns (GH35340)
- Bug in `DataFrame.mode()` and `Series.mode()` not keeping consistent integer `Index` for empty input (GH33321)
- Bug in `DataFrame.rank()` when the `DataFrame` contained `np.inf` (GH32593)
- Bug in `DataFrame.rank()` with `axis=0` and columns holding incomparable types raising an `IndexError` (GH38932)
- Bug in `Series.rank()`, `DataFrame.rank()`, and `GroupBy.rank()` treating the most negative `int64` value as missing (GH32859)
- Bug in `DataFrame.select_dtypes()` different behavior between Windows and Linux with `include="int"` (GH36596)
- Bug in `DataFrame.apply()` and `DataFrame.agg()` when passed the argument `func="size"` would operate on the entire `DataFrame` instead of rows or columns (GH39934)
- Bug in `DataFrame.transform()` would raise a `SpecificationError` when passed a dictionary and columns were missing; will now raise a `KeyError` instead (GH40004)
- Bug in `GroupBy.rank()` giving incorrect results with `pct=True` and equal values between consecutive groups (GH40518)
- Bug in `Series.count()` would result in an `int32` result on 32-bit platforms when argument `level=None` (GH40908)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in `Series` and `DataFrame` reductions with methods `any` and `all` not returning Boolean results for object data (GH12863, GH35450, GH27709)
- Bug in `Series.clip()` would fail if the Series contains NA values and has nullable int or float as a data type (GH40851)
- Bug in `UInt64Index.where()` and `UInt64Index.putmask()` with an np.int64 dtype other incorrectly raising `TypeError` (GH41974)
- Bug in `DataFrame.agg()` not sorting the aggregated axis in the order of the provided aggregation functions when one or more aggregation function fails to produce results (GH33634)
- Bug in `DataFrame.clip()` not interpreting missing values as no threshold (GH40420)

Conversion

- Bug in `Series.to_dict()` with orient='records' now returns Python native types (GH25969)
- Bug in `Series.view()` and `Index.view()` when converting between datetime-like (datetime64[ns], datetime64[ns, tz], timedelta64, period) dtypes (GH39788)
- Bug in creating a `DataFrame` from an empty np.recarray not retaining the original dtypes (GH40121)
- Bug in `DataFrame` failing to raise a `TypeError` when constructing from a `frozenset` (GH40163)
- Bug in `Index` construction silently ignoring a passed dtype when the data cannot be cast to that dtype (GH21311)
- Bug in `StringArray.astype()` falling back to NumPy and raising when converting to dtype='categorical' (GH40450)
- Bug in `factorize()` where, when given an array with a numeric NumPy dtype lower than int64, uint64 and float64, the unique values did not keep their original dtype (GH41132)
- Bug in `DataFrame` construction with a dictionary containing an array-like with `ExtensionDtype` and `copy=True` failing to make a copy (GH38939)
- Bug in `qcut()` raising error when taking Float64DType as input (GH40730)
- Bug in `DataFrame` and `Series` construction with datetime64[ns] data and dtype=object resulting in datetime objects instead of `Timestamp` objects (GH41599)
- Bug in `DataFrame` and `Series` construction with timedelta64[ns] data and dtype=object resulting in np.timedelta64 objects instead of Timedelta objects (GH41599)
- Bug in `DataFrame` construction when given a two-dimensional object-dtype np.ndarray of `Period` or `Interval` objects failing to cast to `PeriodDtype` or `IntervalDtype`, respectively (GH41812)
- Bug in constructing a `Series` from a list and a `PandasDtype` (GH39357)
- Bug in creating a `Series` from a `range` object that does not fit in the bounds of int64 dtype (GH30173)
- Bug in creating a `Series` from a dict with all-tuple keys and an `Index` that requires reindexing (GH1707)
- Bug in `infer_dtype()` not recognizing Series, Index, or array with a Period dtype (GH23553)
- Bug in `infer_dtype()` raising an error for general `ExtensionArray` objects. It will now return "unknown-array" instead of raising (GH37367)
- Bug in `DataFrame.convert_dtypes()` incorrectly raised a `ValueError` when called on an empty DataFrame (GH40393)
pandas: powerful Python data analysis toolkit, Release 1.4.2

Strings

- Bug in the conversion from `pyarrow.ChunkedArray` to `StringArray` when the original had zero chunks (GH41404)
- Bug in `Series.replace()` and `DataFrame.replace()` ignoring replacements with `regex=True` for `StringDType` data (GH41333, GH35977)
- Bug in `Series.str.extract()` with `StringArray` returning object dtype for an empty `DataFrame` (GH41441)
- Bug in `Series.str.replace()` where the `case` argument was ignored when `regex=False` (GH41602)

Interval

- Bug in `IntervalIndex.intersection()` and `IntervalIndex.symmetric_difference()` always returning object-dtype when operating with `CategoricalIndex` (GH38653, GH38741)
- Bug in `IntervalIndex.intersection()` returning duplicates when at least one of the `Index` objects have duplicates which are present in the other (GH38743)
- `IntervalIndex.union()`, `IntervalIndex.intersection()`, `IntervalIndex.difference()`, and `IntervalIndex.symmetric_difference()` now cast to the appropriate dtype instead of raising a `TypeError` when operating with another `IntervalIndex` with incompatible dtype (GH39267)
- `PeriodIndex.union()`, `PeriodIndex.intersection()`, `PeriodIndex.symmetric_difference()`, `PeriodIndex.difference()` now cast to object dtype instead of raising `IncompatibleFrequency` when operating with another `PeriodIndex` with incompatible dtype (GH39306)
- Bug in `IntervalIndex.is_monotonic()`, `IntervalIndex.get_loc()`, `IntervalIndex.get_indexer_for()`, and `IntervalIndex.__contains__()` when NA values are present (GH41831)

Indexing

- Bug in `Index.union()` and `MultiIndex.union()` dropping duplicate `Index` values when `Index` was not monotonic or sort was set to False (GH36289, GH31326, GH40862)
- Bug in `CategoricalIndex.get_indexer()` failing to raise `InvalidIndexError` when non-unique (GH38372)
- Bug in `IntervalIndex.get_indexer()` when target has `CategoricalDtype` and both the index and the target contain NA values (GH41934)
- Bug in `Series.loc()` raising a `ValueError` when input was filtered with a Boolean list and values to set were a list with lower dimension (GH38380)
- Bug in inserting many new columns into a `DataFrame` causing incorrect subsequent indexing behavior (GH15695)
- Bug in `DataFrame.__setitem__()` raising a `ValueError` when setting multiple values to duplicate columns (GH33146)
- Bug in `DataFrame.loc()`, `Series.loc()`, `DataFrame.__getitem__()` and `Series.__getitem__()` returning incorrect elements for non-monotonic `DatetimeIndex` for string slices (GH33146)
- Bug in `DataFrame.reindex()` and `Series.reindex()` with timezone aware indexes raising a `TypeError` for method="ffill" and method="bfill" and specified tolerance (GH38566)
• Bug in DataFrame.reindex() with datetime64[ns] or timedelta64[ns] incorrectly casting to integers when the fill_value requires casting to object dtype (GH39755)

• Bug in DataFrame.__setitem__() raising a ValueError when setting on an empty DataFrame using specified columns and a nonempty DataFrame value (GH38831)

• Bug in DataFrame.loc.__setitem__() raising a ValueError when operating on a unique column when the DataFrame has duplicate columns (GH38521)

• Bug in DataFrame.iloc.__setitem__() and DataFrame.loc.__setitem__() with mixed dtypes when setting with a dictionary value (GH38335)

• Bug in Series.loc.__setitem__() and DataFrame.loc.__setitem__() raising KeyError when provided a Boolean generator (GH39614)

• Bug in Series.iloc() and DataFrame.iloc() raising a KeyError when provided a generator (GH39614)

• Bug in DataFrame.__getitem__() and Series.__getitem__() always raising KeyError when slicing with existing strings where the Index has milliseconds (GH33589)

• Bug in setting timedelta64 or datetime64 values into numeric Series failing to cast to object dtype (GH39086, GH39619)

• Bug in setting Interval values into a Series or DataFrame with mismatched IntervalDtype incorrectly casting the new values to the existing dtype (GH39120)

• Bug in setting datetime64 values into a Series with integer-dtype incorrectly casting the datetime64 values to integers (GH39266)

• Bug in setting np.datetime64("NaT") into a Series with Datetime64TZDtype incorrectly treating the timezone-naive value as timezone-aware (GH39769)

• Bug in Index.get_loc() not raising KeyError when key=NaN and method is specified but NaN is not in the Index (GH39382)

• Bug in DatetimeIndex.insert() when inserting np.datetime64("NaT") into a timezone-aware index incorrectly treating the timezone-naive value as time zone-aware (GH39769)

• Bug in incorrectly raising in Index.insert(), when setting a new column that cannot be held in the existing frame.columns, or in Series.reset_index() or DataFrame.reset_index() instead of casting to a compatible dtype (GH39068)

• Bug in RangeIndex.append() where a single object of length 1 was concatenated incorrectly (GH39401)

• Bug in RangeIndex.astype() where when converting to CategoricalIndex, the categories became a Int64Index instead of a RangeIndex (GH41263)

• Bug in setting numpy.timedelta64 values into an object-dtype Series using a Boolean indexer (GH39488)

• Bug in setting numeric values into a into a boolean-dtypes Series using at or iat failing to cast to object-dtype (GH39582)

• Bug in DataFrame.__setitem__() and DataFrame.iloc.__setitem__() raising ValueError when trying to index with a row-slice and setting a list as values (GH40440)

• Bug in DataFrame.loc() not raising KeyError when the key was not found in MultiIndex and the levels were not fully specified (GH41170)
• Bug in `DataFrame.loc.__setitem__()` when setting-with-expansion incorrectly raising when the index in the expanding axis contained duplicates (GH40096)

• Bug in `DataFrame.loc.__getitem__()` with `MultiIndex` casting to float when at least one index column has float dtype and we retrieve a scalar (GH41369)

• Bug in `DataFrame.loc()` incorrectly matching non-Boolean index elements (GH20432)

• Bug in indexing with `np.nan` on a `Series` or `DataFrame` with a `CategoricalIndex` incorrectly raising `KeyError` when `np.nan` keys are present (GH41933)

• Bug in `Series.__delitem__()` with `ExtensionDtype` incorrectly casting to `ndarray` (GH40386)

• Bug in `DataFrame.at()` with a `CategoricalIndex` returning incorrect results when passed integer keys (GH41846)

• Bug in `DataFrame.loc()` returning a `MultiIndex` in the wrong order if an indexer has duplicates (GH40978)

• Bug in `Index.get_indexer()` failing to raise `ValueError` in some cases with invalid method, limit, or tolerance arguments (GH41918)

• Bug when slicing a `Series` or `DataFrame` with a `TimedeltaIndex` when passing an invalid string raising `ValueError` instead of a `TypeError` (GH41821)

• Bug in Index constructor sometimes silently ignoring a specified `dtype` (GH38879)

• `Index.where()` behavior now mirrors `Index.putmask()` behavior, i.e. `index.where(mask, other)` matches `index.putmask(~mask, other)` (GH39412)

**Missing**

• Bug in `Grouper` did not correctly propagate the `dropna` argument; `DataFrameGroupBy.transform()` now correctly handles missing values for `dropna=True` (GH35612)

• Bug in `isna()`, `Series.isna()`, `Index.isna()`, `DataFrame.isna()`, and the corresponding `notna` functions not recognizing `Decimal("NaN")` objects (GH39409)

• Bug in `DataFrame.fillna()` not accepting a dictionary for the `downcast` keyword (GH40809)

• Bug in `isna()` not returning a copy of the mask for nullable types, causing any subsequent mask modification to change the original array (GH40935)

• Bug in `DataFrame` construction with float data containing `NaN` and an integer `dtype` casting instead of retaining the `NaN` (GH26919)

• Bug in `Series.isin()` and `MultiIndex.isin()` didn’t treat all nans as equivalent if they were in tuples (GH41936)
MultiIndex

- Bug in `DataFrame.drop()` raising a `TypeError` when the `MultiIndex` is non-unique and `level` is not provided (GH36293)
- Bug in `MultiIndex.intersection()` duplicating `NaN` in the result (GH38623)
- Bug in `MultiIndex.equals()` incorrectly returning `True` when the `MultiIndex` contained `NaN` even when they are differently ordered (GH38439)
- Bug in `MultiIndex.intersection()` always returning an empty result when intersecting with `CategoricalIndex` (GH38653)
- Bug in `MultiIndex.difference()` incorrectly raising `TypeError` when indexes contain non-sortable entries (GH41915)
- Bug in `MultiIndex.reindex()` raising a `ValueError` when used on an empty `MultiIndex` and indexing only a specific level (GH41170)
- Bug in `MultiIndex.reindex()` raising `TypeError` when reindexing against a flat `Index` (GH41707)

I/O

- Bug in `Index.__repr__()` when `display.max_seq_items=1` (GH38415)
- Bug in `read_csv()` not recognizing scientific notation if the argument `decimal` is set and `engine="python"` (GH31920)
- Bug in `read_csv()` interpreting `NA` value as comment, when `NA` does contain the comment string fixed for `engine="python"` (GH34002)
- Bug in `read_csv()` raising an `IndexError` with multiple header columns and `index_col` is specified when the file has no data rows (GH38292)
- Bug in `read_csv()` not accepting `usecols` with a different length than `names` for `engine="python"` (GH16469)
- Bug in `read_csv()` returning object dtype when `delimiter=","` with `usecols` and `parse_dates` specified for `engine="python"` (GH35873)
- Bug in `read_csv()` raising a `TypeError` when `names` and `parse_dates` is specified for `engine="c"` (GH33699)
- Bug in `read_clipboard()` and `DataFrame.to_clipboard()` not working in WSL (GH38527)
- Allow custom error values for the `parse_dates` argument of `read_sql()`, `read_sql_query()` and `read_sql_table()` (GH35185)
- Bug in `DataFrame.to_hdf()` and `Series.to_hdf()` raising a `KeyError` when trying to apply for subclasses of DataFrame or Series (GH33748)
- Bug in `HDFStore.put()` raising a wrong `TypeError` when saving a DataFrame with non-string dtype (GH34274)
- Bug in `json_normalize()` resulting in the first element of a generator object not being included in the returned DataFrame (GH35923)
- Bug in `read_csv()` applying the thousands separator to date columns when the column should be parsed for dates and `usecols` is specified for `engine="python"` (GH39365)
- Bug in `read_excel()` forward filling `MultiIndex` names when multiple header and index columns are specified (GH34673)
• Bug in `read_excel()` not respecting `set_option()` (GH34252)
• Bug in `read_csv()` not switching `true_values` and `false_values` for nullable Boolean dtype (GH34655)
• Bug in `read_json()` when `orient="split"` not maintaining a numeric string index (GH28556)
• `read_sql()` returned an empty generator if `chunksize` was non-zero and the query returned no results. Now returns a generator with a single empty DataFrame (GH34411)
• Bug in `read_hdf()` returning unexpected records when filtering on categorical string columns using the `where` parameter (GH39189)
• Bug in `read_sas()` raising a `ValueError` when datetimes were null (GH39725)
• Bug in `read_excel()` dropping empty values from single-column spreadsheets (GH39808)
• Bug in `read_excel()` loading trailing empty rows/columns for some filetypes (GH41167)
• Bug in `read_excel()` raising an `AttributeError` when the excel file had a `MultiIndex` header followed by two empty rows and no index (GH40442)
• Bug in `read_excel()`, `read_csv()`, `read_table()`, `read_fwf()`, and `read_clipboard()` where one blank row after a `MultiIndex` header with no index would be dropped (GH40442)
• Bug in `DataFrame.to_string()` misplacing the truncation column when `index=False` (GH40904)
• Bug in `DataFrame.to_string()` adding an extra dot and misaligning the truncation row when `index=False` (GH40904)
• Bug in `read_orc()` always raising an `AttributeError` (GH40918)
• Bug in `read_csv()` and `read_table()` silently ignoring `prefix` if `names` and `prefix` are defined, now raising a `ValueError` (GH39123)
• Bug in `read_csv()` and `read_excel()` not respecting the dtype for a duplicated column name when `mangle_dupe_cols` is set to True (GH35211)
• Bug in `read_csv()` silently ignoring `sep` if `delimiter` and `sep` are defined, now raising a `ValueError` (GH39823)
• Bug in `read_csv()` and `read_table()` misinterpreting arguments when `sys.setprofile` had been previously called (GH41069)
• Bug in the conversion from PyArrow to pandas (e.g. for reading Parquet) with nullable dtypes and a PyArrow array whose data buffer size is not a multiple of the dtype size (GH40896)
• Bug in `read_excel()` would raise an error when pandas could not determine the file type even though the user specified the `engine` argument (GH4125)
• Bug in `read_clipboard()` copying from an excel file shifts values into the wrong column if there are null values in first column (GH41108)
• Bug in `DataFrame.to_hdf()` and `Series.to_hdf()` raising a `TypeError` when trying to append a string column to an incompatible column (GH41897)
Period

- Comparisons of `Period` objects or `Index`, `Series`, or `DataFrame` with mismatched `PeriodDtype` now behave like other mismatched-type comparisons, returning False for equals, True for not-equal, and raising `TypeError` for inequality checks (GH39274)

Plotting

- Bug in `plotting.scatter_matrix()` raising when 2d ax argument passed (GH16253)
- Prevent warnings when Matplotlib’s `constrained_layout` is enabled (GH25261)
- Bug in `DataFrame.plot()` was showing the wrong colors in the legend if the function was called repeatedly and some calls used `yerr` while others didn’t (GH39522)
- Bug in `DataFrame.plot()` was showing the wrong colors in the legend if the function was called repeatedly and some calls used `secondary_y` and others use `legend=False` (GH40044)
- Bug in `DataFrame.plot.box()` when `dark_background` theme was selected, caps or min/max markers for the plot were not visible (GH40769)

Groupby/resample/rolling

- Bug in `GroupBy.agg()` with `PeriodDtype` columns incorrectly casting results too aggressively (GH38254)
- Bug in `SeriesGroupBy.value_counts()` where unobserved categories in a grouped categorical Series were not tallied (GH38672)
- Bug in `SeriesGroupBy.value_counts()` where an error was raised on an empty Series (GH39172)
- Bug in `GroupBy.indices()` would contain non-existent indices when null values were present in the groupby keys (GH9304)
- Fixed bug in `GroupBy.sum()` causing a loss of precision by now using Kahan summation (GH38778)
- Fixed bug in `GroupBy.cumsum()` and `GroupBy.mean()` causing loss of precision through using Kahan summation (GH38934)
- Bug in `Resampler.aggregate()` and `DataFrame.transform()` raising a `TypeError` instead of `SpecificationError` when missing keys had mixed dtypes (GH39025)
- Bug in `DataFrameGroupBy.idxmin()` and `DataFrameGroupBy.idxmax()` with `ExtensionDtype` columns (GH38733)
- Bug in `Series.resample()` would raise when the index was a `PeriodIndex` consisting of NaT (GH39227)
- Bug in `RollingGroupby.corr()` and `ExpandingGroupby.corr()` where the groupby column would return 0 instead of `np.nan` when providing `other` that was longer than each group (GH39591)
- Bug in `ExpandingGroupby.corr()` and `ExpandingGroupby.cov()` where 1 would be returned instead of `np.nan` when providing `other` that was longer than each group (GH39591)
- Bug in `GroupBy.mean()`, `GroupBy.median()` and `DataFrame.pivot_table()` not propagating metadata (GH28283)
- Bug in `Series.rolling()` and `DataFrame.rolling()` not calculating window bounds correctly when window is an offset and dates are in descending order (GH40002)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in `Series.groupby()` and `DataFrame.groupby()` on an empty Series or DataFrame would lose index, columns, and/or data types when directly using the methods `idxmax`, `idxmin`, `mad`, `min`, `max`, `sum`, `prod`, and `skew` or using them through `apply`, `aggregate`, or `resample` (GH26411)
- Bug in `GroupBy.apply()` where a `MultiIndex` would be created instead of an `Index` when used on a `RollingGroupby` object (GH39732)
- Bug in `DataFrameGroupBy.sample()` where an error was raised when `weights` was specified and the index was an `Int64Index` (GH39927)
- Bug in `DataFrameGroupBy.aggregate()` and `Resampler.aggregate()` would sometimes raise a `SpecificationError` when passed a dictionary and columns were missing; will now always raise a `KeyError` instead (GH40004)
- Bug in `DataFrameGroupBy.sample()` where column selection was not applied before computing the result (GH39928)
- Bug in `ExponentialMovingWindow` when calling `__getitem__` would incorrectly raise a `ValueError` when providing times (GH40164)
- Bug in `ExponentialMovingWindow` when calling `__getitem__` would not retain `com`, `span`, `alpha` or `halflife` attributes (GH40164)
- `ExponentialMovingWindow` now raises a `NotImplementedError` when specifying times with `adjust=False` due to an incorrect calculation (GH40098)
- Bug in `ExponentialMovingWindowGroupby.mean()` where the times argument was ignored when `engine='numba'` (GH40951)
- Bug in `ExponentialMovingWindowGroupby.mean()` where the wrong times were used the in case of multiple groups (GH40951)
- Bug in `ExponentialMovingWindowGroupby` where the times vector and values became out of sync for non-trivial groups (GH40951)
- Bug in `Series.asfreq()` and `DataFrame.asfreq()` dropping rows when the index was not sorted (GH39805)
- Bug in aggregation functions for `DataFrame` not respecting `numeric_only` argument when `level` keyword was given (GH40660)
- Bug in `SeriesGroupBy.aggregate()` where using a user-defined function to aggregate a Series with an object-typed `Index` causes an incorrect `Index` shape (GH40014)
- Bug in `RollingGroupby` where `as_index=False` argument in `groupby` was ignored (GH39433)
- Bug in `GroupBy.any()` and `GroupBy.all()` raising a `ValueError` when using with nullable type columns holding `NA` even with `skipna=True` (GH40585)
- Bug in `GroupBy.cummin()` and `GroupBy.cummax()` incorrectly rounding integer values near the int64 implementations bounds (GH40767)
- Bug in `GroupBy.rank()` with nullable dtypes incorrectly raising a `TypeError` (GH41010)
- Bug in `GroupBy.cummin()` and `GroupBy.cummax()` computing wrong result with nullable data types too large to roundtrip when casting to float (GH37493)
- Bug in `DataFrame.rolling()` returning mean zero for all NaN window with `min_periods=0` if calculation is not numerical stable (GH41053)
- Bug in `DataFrame.rolling()` returning sum not zero for all NaN window with `min_periods=0` if calculation is not numerical stable (GH41053)
- Bug in `SeriesGroupBy.agg()` failing to retain ordered `CategoricalDtype` on order-preserving aggregations (GH41147)
• Bug in `GroupBy.min()` and `GroupBy.max()` with multiple object-dtype columns and `numeric_only=False` incorrectly raising a `ValueError` (GH41111)

• Bug in `DataFrameGroupBy.rank()` with the GroupBy object’s `axis=0` and the `rank` method’s keyword `axis=1` (GH41320)

• Bug in `DataFrameGroupBy.__getitem__()` with non-unique columns incorrectly returning a malformed `SeriesGroupBy` instead of `DataFrameGroupBy` (GH41427)

• Bug in `DataFrameGroupBy.transform()` with non-unique columns incorrectly raising an `AttributeError` (GH41427)

• Bug in `Resampler.apply()` with non-unique columns incorrectly dropping duplicated columns (GH41445)

• Bug in `Series.groupby()` aggregations incorrectly returning empty `Series` instead of raising `TypeError` on aggregations that are invalid for its dtype, e.g. `.prod` with `datetime64[ns]` dtype (GH41342)

• Bug in `DataFrameGroupBy` aggregations incorrectly failing to drop columns with invalid dtypes for that aggregation when there are no valid columns (GH41291)

• Bug in `DataFrame.rolling.__iter__()` where `on` was not assigned to the index of the resulting objects (GH40373)

• Bug in `DataFrameGroupBy.transform()` and `DataFrameGroupBy.agg()` with `engine="numba"` where *args were being cached with the user passed function (GH41647)

• Bug in `DataFrameGroupBy` methods `agg`, `transform`, `sum`, `bfill`, `ffill`, `pad`, `pct_change`, `shift`, `ohlc` dropping `.columns.names` (GH41497)

Reshaping

• Bug in `merge()` raising error when performing an inner join with partial index and `right_index=True` when there was no overlap between indices (GH33814)

• Bug in `DataFrame.unstack()` with missing levels led to incorrect index names (GH37510)

• Bug in `merge.asof()` propagating the right Index with `left_index=True` and `right_on` specification instead of left Index (GH33463)

• Bug in `DataFrame.join()` on a DataFrame with a `MultiIndex` returned the wrong result when one of both indexes had only one level (GH36909)

• `merge.asof()` now raises a `ValueError` instead of a cryptic `TypeError` in case of non-numerical merge columns (GH29130)

• Bug in `DataFrame.join()` not assigning values correctly when the DataFrame had a `MultiIndex` where at least one dimension had dtype Categorical with non-alphabetically sorted categories (GH38502)

• `Series.value_counts()` and `Series.mode()` now return consistent keys in original order (GH12679, GH11227 and GH39007)

• Bug in `DataFrame.stack()` not handling NaN in `MultiIndex` columns correctly (GH39481)

• Bug in `DataFrame.apply()` would give incorrect results when the argument `func` was a string, `axis=1`, and the axis argument was not supported; now raises a `ValueError` instead (GH39211)

• Bug in `DataFrame.sort_values()` not reshaping the index correctly after sorting on columns when `ignore_index=True` (GH39464)

• Bug in `DataFrame.append()` returning incorrect dtypes with combinations of `ExtensionDtype` dtypes (GH39454)

5.2. Version 1.3 2931
- Bug in `DataFrame.append()` returning incorrect dtypes when used with combinations of `datetime64` and `timedelta64` dtypes (GH39574)
- Bug in `DataFrame.append()` with a `DataFrame` with a `MultiIndex` and appending a `Series` whose `Index` is not a `MultiIndex` (GH41707)
- Bug in `DataFrame.pivot_table()` returning a `MultiIndex` for a single value when operating on an empty `DataFrame` (GH13483)
- `Index` can now be passed to the `numpy.all()` function (GH40180)
- Bug in `DataFrame.stack()` not preserving `CategoricalDtype` in a `MultiIndex` (GH36991)
- Bug in `to_datetime()` raising an error when the input sequence contained unhashable items (GH39756)
- Bug in `Series.explode()` preserving the index when `ignore_index` was `True` and values were scalars (GH40487)
- Bug in `to_datetime()` raising a `ValueError` when `Series` contains `None` and `NaT` and has more than 50 elements (GH39882)
- Bug in `Series.unstack()` and `DataFrame.unstack()` with object-dtype values containing timezone-aware datetime objects incorrectly raising `TypeError` (GH41875)
- Bug in `DataFrame.melt()` raising `InvalidIndexError` when `DataFrame` has duplicate columns used as `value_vars` (GH41951)

**Sparse**

- Bug in `DataFrame.sparse.to_coo()` raising a `KeyError` with columns that are a numeric `Index` without a `0` (GH18414)
- Bug in `SparseArray.astype()` with `copy=False` producing incorrect results when going from integer dtype to floating dtype (GH34456)
- Bug in `SparseArray.max()` and `SparseArray.min()` would always return an empty result (GH40921)

**ExtensionArray**

- Bug in `DataFrame.where()` when `other` is a Series with an `ExtensionDtype` (GH38729)
- Fixed bug where `Series.idxmax()`, `Series.idxmin()`, `Series.argmax()`, and `Series.argmin()` would fail when the underlying data is an `ExtensionArray` (GH32749, GH33719, GH36566)
- Fixed bug where some properties of subclasses of `PandasExtensionDtype` where improperly cached (GH40329)
- Bug in `DataFrame.mask()` where masking a `DataFrame` with an `ExtensionDtype` raises a `ValueError` (GH40941)
Styler

- Bug in Styler where the subset argument in methods raised an error for some valid MultiIndex slices (GH33562)
- Styler rendered HTML output has seen minor alterations to support w3 good code standards (GH39626)
- Bug in Styler where rendered HTML was missing a column class identifier for certain header cells (GH39716)
- Bug in Styler.background_gradient() where text-color was not determined correctly (GH39888)
- Bug in Styler.set_table_styles() where multiple elements in CSS-selectors of the table_styles argument were not correctly added (GH34061)
- Bug in Styler where copying from Jupyter dropped the top left cell and misaligned headers (GH12147)
- Bug in Styler.where where kwargs were not passed to the applicable callable (GH40845)
- Bug in Styler causing CSS to duplicate on multiple renders (GH39395, GH40334)

Other

- inspect.getmembers(Series) no longer raises an AbstractMethodError (GH38782)
- Bug in Series.where() with numeric dtype and other=None not casting to nan (GH39761)
- Bug in assert_series_equal(), assert_frame_equal(), assert_index_equal() and assert_extension_array_equal() incorrectly raising when an attribute has an unrecognized NA type (GH39461)
- Bug in assert_index_equal() with exact=True not raising when comparing CategoricalIndex instances with Int64Index and RangeIndex categories (GH14263)
- Bug in DataFrame.equals(), Series.equals(), and Index.equals() with object-dtype containing np.datetime64("NaT") or np.timedelta64("NaT") (GH39650)
- Bug in show_versions() where console JSON output was not proper JSON (GH39701)
- pandas can now compile on z/OS when using xlc (GH35826)
- Bug in pandas.util.hash_pandas_object() not recognizing hash_key, encoding and categorize when the input object type is a DataFrame (GH41404)

Contributors

A total of 251 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Abhishek R +
- Ada Draginda
- Adam J. Stewart
- Adam Turner +
- Aidan Feldman +
- Ajitesh Singh +
- Akshat Jain +
- Albert Villanova del Moral
• Alexandre Prince-Levasseur +
• Andrew Hawyrluk +
• Andrew Wieteska
• AnglinaBhambra +
• Ankush Dua +
• Anna Daglis
• Ashlan Parker +
• Ashwani +
• Avinash Pancham
• Ayushman Kumar +
• BeanNan
• Benoît Vinot
• Bharat Raghunathan
• Bijay Regmi +
• Bobin Mathew +
• Bogdan Pilyavets +
• Brian Hulette +
• Brian Sun +
• Brock +
• Bryan Cutler
• Caleb +
• Calvin Ho +
• Chathura Widanage +
• Chinmay Rane +
• Chris Lynch
• Chris Withers
• Christos Petropoulos
• Corentin Girard +
• DaPy15 +
• Damodara Puddu +
• Daniel Hrisca
• Daniel Saxton
• DanielFEvans
• Dare Adewumi +
• Dave Willmer
• David Schlachter +
• David-dmh +
• Deepang Raval +
• Doris Lee +
• Dr. Jan-Philip Gehrcke +
• DriesS +
• Dylan Percy
• Erfan Nariman
• Eric Leung
• EricLeer +
• Eve
• Fangchen Li
• Felix Divo
• Florian Jetter
• Fred Reiss
• GFJ138 +
• Gaurav Sheni +
• Geoffrey B. Eisenbarth +
• Gesa Stupperich +
• Griffin Ansel +
• Gustavo C. Maciel +
• Heidi +
• Henry +
• Hung-Yi Wu +
• Ian Ozsvald +
• Irv Lustig
• Isaac Chung +
• Isaac Virshup
• JHM Darbyshire (MBP) +
• JHM Darbyshire (iMac) +
• Jack Liu +
• James Lamb +
• Jeet Parekh
• Jeff Reback
• Jiezheng2018 +
• Jody Klymak
• Johan Kährström +
• John McGuigan
• Joris Van den Bossche
• Jose
• JoseNavy
• Josh Dimarsky
• Josh Friedlander
• Joshua Klein +
• Julia Signell
• Julian Schnitzler +
• Kaiqi Dong
• Kasim Panjri +
• Katie Smith +
• Kelly +
• Kenil +
• Keppler, Kyle +
• Kevin Sheppard
• Khor Chean Wei +
• Kiley Hewitt +
• Larry Wong +
• Lightyears +
• Lucas Holtz +
• Lucas Rodés-Guirao
• Lucky Sivagurunathan +
• Luis Pinto
• Maciej Kos +
• Marc Garcia
• Marco Edward Gorelli +
• Marco Gorelli
• MarcoGorelli +
• Mark Graham
• Martin Dengler +
• Martin Grigorov +
• Marty Rudolf +
• Matt Roeschke
• Matthew Roeschke
• Matthew Zeitlin
• Max Bolingbroke
• Maxim Ivanov
• Maxim Kupfer +
• Mayur +
• MeeseeksMachine
• Micael Jarniac
• Michael Hsieh +
• Michel de Ruiter +
• Mike Roberts +
• Miroslav Šedivý
• Mohammad Jafar Mashhadi
• Morisa Manzella +
• Mortada Mehyar
• Muktan +
• Naveen Agrawal +
• Noah
• Nofar Mishraki +
• Oleh Kozynets
• Olga Matoula +
• Oli +
• Omar Afifi
• Omer Ozarslan +
• Owen Lamont +
• Ozan Öğreden +
• Pandas Development Team
• Paolo Lammens
• Parfait Gasana +
• Patrick Hoefler
• Paul McCarthy +
• Paulo S. Costa +
• Pav A
• Peter
• Pradyumna Rahul +
• Punitvara +
• QP Hou +
• Rahul Chauhan
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Rahul Sathanapalli
- Richard Shadrach
- Robert Bradshaw
- Robin to Roxel
- Rohit Gupta
- Sam Purkis +
- Samuel GIFFARD +
- Sean M. Law +
- Shahar Naveh +
- ShaharNaveh +
- Shiv Gupta +
- Shrey Dixit +
- Shudong Yang +
- Simon Boehm +
- Simon Hawkins
- Sioned Baker +
- Stefan Mejlaaard +
- Steven Pitman +
- Steven Schaefer +
- Stéphane Guillou +
- TLouf +
- Tegar D Pratama +
- Terji Petersen
- Theodoros Nikolaou +
- Thomas Dickson
- Thomas Li
- Thomas Smith
- Thomas Yu +
- ThomasBlauthQC +
- Tim Hoffmann
- Tom Augspurger
- Torsten Wörtwein
- Tyler Reddy
- UrielMaD
- Uwe L. Korn
- Venaturum +
• VirosaLi
• Vladimir Podolskiy
• Vyom Pathak +
• WANG Aiyong
• Waltteri Koskinen +
• Wenjun Si +
• William Ayd
• Yeshwanth N +
• Yuanhao Geng
• Zito Relova +
• aflah02 +
• arredond +
• attack68
• cdknox +
• chinggg +
• fathomer +
• ftrihardjo +
• github-actions[bot] +
• gunjan-solanki +
• guru kiran
• hasan-yaman
• i-aki-y +
• jbrockmendel
• jmholzer +
• jordi-crespo +
• jotasi +
• jreback
• juliansmidek +
• kylekeppler
• lrepiton +
• lucasrodes
• maroth96 +
• mikeronayne +
• mlondschen
• moink +
• morrme
5.3 Version 1.2

5.3.1 What’s new in 1.2.5 (June 22, 2021)

These are the changes in pandas 1.2.5. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Fixed regression in `concat()` between two `DataFrame` where one has an `Index` that is all-None and the other is `DatetimeIndex` incorrectly raising (GH40841)
- Fixed regression in `DataFrame.sum()` and `DataFrame.prod()` when `min_count` and `numeric_only` are both given (GH41074)
- Fixed regression in `read_csv()` when using `memory_map=True` with a non-UTF8 encoding (GH40986)
- Fixed regression in `DataFrame.replace()` and `Series.replace()` when the values to replace is a NumPy float array (GH40371)
- Fixed regression in `ExcelFile()` when a corrupt file is opened but not closed (GH41778)
- Fixed regression in `DataFrame.astype()` with `dtype=str` failing to convert NaN in categorical columns (GH41797)

Contributors

A total of 12 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Joris Van den Bossche
- Marc Garcia
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Patrick Hoefler +
- Simon Hawkins
- Thomas Li
- Torsten Wörtwein
- hasan-yaman +
- jbrockmendel
- phofl +

5.3.2 What’s new in 1.2.4 (April 12, 2021)

These are the changes in pandas 1.2.4. See Release notes for a full changelog including other versions of pandas.
Fixed regressions

- Fixed regression in `DataFrame.sum()` when `min_count` greater than the `DataFrame` shape was passed resulted in a `ValueError` (GH39738)
- Fixed regression in `DataFrame.to_json()` raising `AttributeError` when run on PyPy (GH39837)
- Fixed regression in (in)equality comparison of `pd.NaT` with a non-datetime-like numpy array returning a scalar instead of an array (GH40722)
- Fixed regression in `DataFrame.where()` not returning a copy in the case of an all True condition (GH39595)
- Fixed regression in `DataFrame.replace()` raising `IndexError` when `regex` was a multi-key dictionary (GH39338)
- Fixed regression in repr of floats in an `object` column not respecting `float_format` when printed in the console or outputted through `DataFrame.to_string()`, `DataFrame.to_html()`, and `DataFrame.to_latex()` (GH40024)
- Fixed regression in NumPy ufuncs such as `np.add` not passing through all arguments for `DataFrame` (GH40662)

Contributors

A total of 9 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Fangchen Li
- Joris Van den Bossche
- Matthew Zeitlin
- MeeseeksMachine
- Pandas Development Team
- Simon Hawkins
- jbrockmendel
- mzeitlin11
- patrick

5.3.3 What’s new in 1.2.3 (March 02, 2021)

These are the changes in pandas 1.2.3. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `to_excel()` raising `KeyError` when giving duplicate columns with `columns` attribute (GH39695)
- Fixed regression in nullable integer unary ops propagating mask on assignment (GH39943)
- Fixed regression in `DataFrame.__setitem__()` not aligning `DataFrame` on right-hand side for boolean indexer (GH39931)
- Fixed regression in `to_json()` failing to use compression with URL-like paths that are internally opened in binary mode or with user-provided file objects that are opened in binary mode (GH39985)
• Fixed regression in `Series.sort_index()` and `DataFrame.sort_index()`, which exited with an ungraceful error when having kwarg `ascending=None` passed. Passing `ascending=None` is still considered invalid, and the improved error message suggests a proper usage (`ascending` must be a boolean or a list-like of boolean) (GH39434)

• Fixed regression in `DataFrame.transform()` and `Series.transform()` giving incorrect column labels when passed a dictionary with a mix of list and non-list values (GH40018)

**Contributors**

A total of 14 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Daniel Saxton
- Dylan Percy +
- Jody Klymak +
- Maxim Ivanov
- MeeseeksMachine
- Pandas Development Team
- Richard Shadrach
- Rohit Gupta +
- Simon Hawkins
- Torsten Wörtwein
- jbrockmendel
- morrme +
- partev
- patrick

**5.3.4 What’s new in 1.2.2 (February 09, 2021)**

These are the changes in pandas 1.2.2. See Release notes for a full changelog including other versions of pandas.

**Fixed regressions**

• Fixed regression in `read_excel()` that caused it to raise `AttributeError` when checking version of older `xlrd` versions (GH38955)

• Fixed regression in `DataFrame` constructor reordering element when construction from datetime ndarray with dtype not "datetime64[ns]" (GH39422)

• Fixed regression in `DataFrame.astype()` and `Series.astype()` not casting to bytes dtype (GH39474)

• Fixed regression in `to_pickle()` failing to create bz2/xz compressed pickle files with protocol=5 (GH39002)

• Fixed regression in `pandas.testing.assert_series_equal()` and `pandas.testing.assert_frame_equal()` always raising `AssertionError` when comparing extension dtypes (GH39410)

• Fixed regression in `to_csv()` opening `codecs.StreamWriter` in binary mode instead of in text mode and ignoring user-provided mode (GH39247)
• Fixed regression in `Categorical.astype()` casting to incorrect dtype when np.int32 is passed to dtype argument (GH39402)
• Fixed regression in `to_excel()` creating corrupt files when appending (mode="a") to an existing file (GH39576)
• Fixed regression in `DataFrame.transform()` failing in case of an empty DataFrame or Series (GH39636)
• Fixed regression in `groupby()` or `resample()` when aggregating an all-NaN or numeric object dtype column (GH39329)
• Fixed regression in `Rolling.count()` where the `min_periods` argument would be set to 0 after the operation (GH39554)
• Fixed regression in `read_excel()` that incorrectly raised when the argument `io` was a non-path and non-buffer and the `engine` argument was specified (GH39528)

Bug fixes

• `pandas.read_excel()` error message when a specified `sheetname` does not exist is now uniform across engines (GH39250)
• Fixed bug in `pandas.read_excel()` producing incorrect results when the engine `openpyxl` is used and the excel file is missing or has incorrect dimension information; the fix requires `openpyxl` >= 3.0.0, prior versions may still fail (GH38956, GH39001)
• Fixed bug in `pandas.read_excel()` sometimes producing a DataFrame with trailing rows of `np.nan` when the engine `openpyxl` is used (GH39181)

Contributors

A total of 14 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Andrew Wieteska
• Erfan Nariman
• Joris Van den Bossche
• Matthew Roeschke
• MeeseeksMachine
• Pandas Development Team
• Pav A
• Richard Shadrach
• Simon Hawkins
• Thomas Li
• Torsten Wörtwein
• Vladimir Podolskiy +
• jbrockmendel
• patrick
5.3.5 What's new in 1.2.1 (January 20, 2021)

These are the changes in pandas 1.2.1. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `to_csv()` that created corrupted zip files when there were more rows than chunksize (GH38714)
- Fixed regression in `to_csv()` opening codecs.StreamReaderWriter in binary mode instead of in text mode (GH39247)
- Fixed regression in `read_csv()` and other read functions were the encoding error policy (errors) did not default to "replace" when no encoding was specified (GH38989)
- Fixed regression in `read_excel()` with non-rawbyte file handles (GH38788)
- Fixed regression in `DataFrame.to_stata()` not removing the created file when an error occurred (GH39202)
- Fixed regression in `DataFrame.__setitem__` raising `ValueError` when expanding `DataFrame` and new column is from type "'0 - name" (GH39010)
- Fixed regression in setting with `DataFrame.loc()` raising `ValueError` when `DataFrame` has unsorted `MultiIndex` columns and indexer is a scalar (GH38601)
- Fixed regression in setting with `DataFrame.loc()` raising `KeyError` with `MultiIndex` and list-like columns indexer enlarging `DataFrame` (GH39147)
- Fixed regression in `groupby()` with `Categorical` grouping column not showing unused categories for `grouped.indices` (GH38642)
- Fixed regression in `GroupBy.sem()` where the presence of non-numeric columns would cause an error instead of being dropped (GH38774)
- Fixed regression in `DataFrameGroupBy.diff()` raising for int8 and int16 columns (GH39050)
- Fixed regression in `DataFrame.groupby()` when aggregating an ExtensionDType that could fail for non-numeric values (GH38980)
- Fixed regression in `Rolling.skew()` and `Rolling.kurt()` modifying the object inplace (GH38908)
- Fixed regression in `DataFrame.any()` and `DataFrame.all()` not returning a result for tz-aware datetime64 columns (GH38723)
- Fixed regression in `DataFrame.apply()` with `axis=1` using str accessor in apply function (GH38979)
- Fixed regression in `DataFrame.replace()` raising `ValueError` when `DataFrame` has dtype bytes (GH38900)
- Fixed regression in `Series.fillna()` that raised `RecursionError` with datetime64[ns, UTC] dtype (GH38851)
- Fixed regression in comparisons between `NaT` and `datetime.date` objects incorrectly returning `True` (GH39151)
- Fixed regression in calling NumPy accumulate() ufuncs on DataFrames, e.g. `np.maximum.accumulate(df)` (GH39259)
- Fixed regression in repr of float-like strings of an object dtype having trailing 0's truncated after the decimal (GH38708)
- Fixed regression that raised `AttributeError` with PyArrow versions [0.16.0, 1.0.0) (GH38801)
• Fixed regression in `pandas.testing.assert_frame_equal()` raising `TypeError` with `check_like=True` when `Index` or columns have mixed dtype (GH39168)

We have reverted a commit that resulted in several plotting related regressions in pandas 1.2.0 (GH38969, GH38736, GH38865, GH38947 and GH39126). As a result, bugs reported as fixed in pandas 1.2.0 related to inconsistent tick labeling in bar plots are again present (GH26186 and GH11465)

**Calling NumPy ufuncs on non-aligned DataFrames**

Before pandas 1.2.0, calling a NumPy ufunc on non-aligned DataFrames (or DataFrame / Series combination) would ignore the indices, only match the inputs by shape, and use the index/columns of the first DataFrame for the result:

```
In [1]: df1 = pd.DataFrame({"a": [1, 2], "b": [3, 4]}, index=[0, 1])
In [2]: df2 = pd.DataFrame({"a": [1, 2], "b": [3, 4]}, index=[1, 2])
In [3]: df1
Out[3]:
   a  b
0  1  3
1  2  4
In [4]: df2
Out[4]:
   a  b
1  1  3
2  2  4
In [5]: np.add(df1, df2)
Out[5]:
   a  b
0 NaN NaN
1  3.0  7.0
2 NaN NaN
```

This contrasts with how other pandas operations work, which first align the inputs:

```
In [6]: df1 + df2
Out[6]:
   a  b
0 NaN NaN
1  3.0  7.0
2 NaN NaN
```

In pandas 1.2.0, we refactored how NumPy ufuncs are called on DataFrames, and this started to align the inputs first (GH39184), as happens in other pandas operations and as it happens for ufuncs called on Series objects.

For pandas 1.2.1, we restored the previous behaviour to avoid a breaking change, but the above example of `np.add(df1, df2)` with non-aligned inputs will now to raise a warning, and a future pandas 2.0 release will start aligning the inputs first (GH39184). Calling a NumPy ufunc on Series objects (eg `np.add(s1, s2)`) already aligns and continues to do so.

To avoid the warning and keep the current behaviour of ignoring the indices, convert one of the arguments to a NumPy array:

```
In [7]: np.add(df1, np.asarray(df2))
Out[7]:
   a  b
0  2  6
1  4  8
```

(continues on next page)
To obtain the future behaviour and silence the warning, you can align manually before passing the arguments to the ufunc:

```python
In [8]: df1, df2 = df1.align(df2)
In [9]: np.add(df1, df2)
Out[9]:
   a   b
0  NaN  NaN
1  3.0  7.0
2  NaN  NaN
```

### Bug fixes

- Bug in `read_csv()` with `float_precision="high"` caused segfault or wrong parsing of long exponent strings. This resulted in a regression in some cases as the default for `float_precision` was changed in pandas 1.2.0 (GH38753)
- Bug in `read_csv()` not closing an opened file handle when a `csv.Error` or `UnicodeDecodeError` occurred while initializing (GH39024)
- Bug in `pandas.testing.assert_index_equal()` raising `TypeError` with `check_order=False` when `Index` has mixed dtype (GH39168)

### Other

- The deprecated attributes `_AXIS_NAMES` and `_AXIS_NUMBERS` of `DataFrame` and `Series` will no longer show up in `dir` or `inspect.getmembers` calls (GH38740)
- Bumped minimum fastparquet version to 0.4.0 to avoid `AttributeError` from numba (GH38344)
- Bumped minimum pymysql version to 0.8.1 to avoid test failures (GH38344)
- Fixed build failure on MacOS 11 in Python 3.9.1 (GH38766)
- Added reference to backwards incompatible `check_freq` arg of `testing.assert_frame_equal()` and `testing.assert_series_equal()` in `pandas 1.1.0 what's new` (GH34050)

### Contributors

A total of 20 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Ada Draginda +
- Andrew Wieteska
- Bryan Cutler
- Fangchen Li
- Joris Van den Bossche
- Matthew Roeschke
5.3.6 What’s new in 1.2.0 (December 26, 2020)

These are the changes in pandas 1.2.0. See Release notes for a full changelog including other versions of pandas.

Warning: The xlwt package for writing old-style .xls excel files is no longer maintained. The xlrd package is now only for reading old-style .xls files.

Previously, the default argument engine=None to read_excel() would result in using the xlrd engine in many cases, including new Excel 2007+ (.xlsx) files. If openpyxl is installed, many of these cases will now default to using the openpyxl engine. See the read_excel() documentation for more details.

Thus, it is strongly encouraged to install openpyxl to read Excel 2007+ (.xlsx) files. Please do not report issues when using `xlrd` to read `.xlsx` files. This is no longer supported, switch to using openpyxl instead.

Attempting to use the xlwt engine will raise a FutureWarning unless the option io.excel.xls.writer is set to "xlwt". While this option is now deprecated and will also raise a FutureWarning, it can be globally set and the warning suppressed. Users are recommended to write .xlsx files using the openpyxl engine instead.

Enhancements

Optionally disallow duplicate labels

Series and DataFrame can now be created with allows_duplicate_labels=False flag to control whether the index or columns can contain duplicate labels (GH28394). This can be used to prevent accidental introduction of duplicate labels, which can affect downstream operations.

By default, duplicates continue to be allowed.
pandas: powerful Python data analysis toolkit, Release 1.4.2

```python
In [1]: pd.Series([1, 2], index=['a', 'a'])
Out[1]:
a 1
a 2
Length: 2, dtype: int64

In [2]: pd.Series([1, 2], index=['a', 'a']).set_flags(allow_duplicates_labels=False)
...: DuplicateLabelError: Index has duplicates.
    positions
    label
    a          [0, 1]
```

pandas will propagate the `allow_duplicates_labels` property through many operations.

```python
In [3]: a = (...
   ...:     pd.Series([1, 2], index=['a', 'b'])
   ...:     .set_flags(allow_duplicates_labels=False)
   ...: )

In [4]: a
Out[4]:
a 1
b 2
Length: 2, dtype: int64

# An operation introducing duplicates
In [5]: a.reindex(['a', 'b', 'a'])
...: DuplicateLabelError: Index has duplicates.
    positions
    label
    a          [0, 2]
```

**Warning:** This is an experimental feature. Currently, many methods fail to propagate the `allow_duplicates_labels` value. In future versions it is expected that every method taking or returning one or more DataFrame or Series objects will propagate `allow_duplicates_labels`.

See *Duplicate Labels* for more.

The `allow_duplicates_labels` flag is stored in the new `DataFrame.flags` attribute. This stores global attributes that apply to the pandas object. This differs from `DataFrame.attrs`, which stores information that applies to the dataset.
Passing arguments to fsspec backends

Many read/write functions have acquired the storage_options optional argument, to pass a dictionary of parameters to the storage backend. This allows, for example, for passing credentials to S3 and GCS storage. The details of what parameters can be passed to which backends can be found in the documentation of the individual storage backends (detailed from the fsspec docs for builtin implementations and linked to external ones). See Section Reading/writing remote files.

GH35655 added fsspec support (including storage_options) for reading excel files.

Support for binary file handles in to_csv

to_csv() supports file handles in binary mode (GH19827 and GH35058) with encoding (GH13068 and GH23854) and compression (GH22555). If pandas does not automatically detect whether the file handle is opened in binary or text mode, it is necessary to provide mode="wb".

For example:

```python
In [1]: import io
In [2]: data = pd.DataFrame([0, 1, 2])
In [3]: buffer = io.BytesIO()
In [4]: data.to_csv(buffer, encoding="utf-8", compression="gzip")
```

Support for short caption and table position in to_latex

DataFrame.to_latex() now allows one to specify a floating table position (GH35281) and a short caption (GH36267).

The keyword position has been added to set the position.

```python
In [5]: data = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
In [6]: table = data.to_latex(position='ht')

In [7]: print(table)
\begin{table}[ht]
\centering
\begin{tabular}{lrr}
\toprule
{} & a & b \\
\midrule
0 & 1 & 3 \\
1 & 2 & 4 \\
\bottomrule
\end{tabular}
\end{table}
```

Usage of the keyword caption has been extended. Besides taking a single string as an argument, one can optionally provide a tuple (full_caption, short_caption) to add a short caption macro.
In [8]: data = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})

In [9]: table = data.to_latex(caption=('the full long caption', 'short caption'))

In [10]: print(table)
\begin{table}
\centering
\caption[short caption]{the full long caption}
\begin{tabular}{lrr}
\toprule
{} & a & b \\
\midrule
0 & 1 & 3 \\
1 & 2 & 4 \\
\bottomrule
\end{tabular}
\end{table}

Change in default floating precision for read_csv and read_table

For the C parsing engine, the methods `read_csv()` and `read_table()` previously defaulted to a parser that could read floating point numbers slightly incorrectly with respect to the last bit in precision. The option `floating_precision=\textit{"high"}` has always been available to avoid this issue. Beginning with this version, the default is now to use the more accurate parser by making `floating_precision=None` correspond to the high precision parser, and the new option `floating_precision="legacy"` to use the legacy parser. The change to using the higher precision parser by default should have no impact on performance. (GH17154)

Experimental nullable data types for float data

We’ve added `Float32Dtype` / `Float64Dtype` and `FloatingArray`. These are extension data types dedicated to floating point data that can hold the `pd.NA` missing value indicator (GH32265, GH34307).

While the default float data type already supports missing values using `np.nan`, these new data types use `pd.NA` (and its corresponding behavior) as the missing value indicator, in line with the already existing nullable `integer` and `boolean` data types.

One example where the behavior of `np.nan` and `pd.NA` is different is comparison operations:

```
# the default NumPy float64 dtype
In [11]: s1 = pd.Series([1.5, None])

In [12]: s1 > 1
Out[12]:
0  True
1  False
dtype: bool
```
# the new nullable float64 dtype
In [14]: s2 = pd.Series([1.5, None], dtype="Float64")

In [15]: s2
Out[15]:
0   1.5
1  <NA>
dtype: Float64

In [16]: s2 > 1
Out[16]:
0   True
1  <NA>
dtype: boolean

See the Experimental NA scalar to denote missing values doc section for more details on the behavior when using the pd.NA missing value indicator.

As shown above, the dtype can be specified using the “Float64” or “Float32” string (capitalized to distinguish it from the default “float64” data type). Alternatively, you can also use the dtype object:

In [17]: pd.Series([1.5, None], dtype=pd.Float32Dtype())
Out[17]:
0   1.5
1  <NA>
dtype: Float32

Operations with the existing integer or boolean nullable data types that give float results will now also use the nullable floating data types (GH38178).

**Warning:** Experimental: the new floating data types are currently experimental, and their behavior or API may still change without warning. Especially the behavior regarding NaN (distinct from NA missing values) is subject to change.

**Index/column name preservation when aggregating**

When aggregating using concat() or the DataFrame constructor, pandas will now attempt to preserve index and column names whenever possible (GH35847). In the case where all inputs share a common name, this name will be assigned to the result. When the input names do not all agree, the result will be unnamed. Here is an example where the index name is preserved:

In [18]: idx = pd.Index(range(5), name='abc')

In [19]: ser = pd.Series(range(5, 10), index=idx)

In [20]: pd.concat({'x': ser[1:], 'y': ser[:-1]}, axis=1)
Out[20]:
     x  y
abc  
1  6.0  6.0
2  7.0  7.0

(continues on next page)
The same is true for `MultiIndex`, but the logic is applied separately on a level-by-level basis.

**GroupBy supports EWM operations directly**

DataframeGroupBy now supports exponentially weighted window operations directly (GH16037).

```python
In [21]: df = pd.DataFrame({'A': ['a', 'b', 'a', 'b'], 'B': range(4)})

In [22]: df
Out[22]:
   A  B
0  a  0
1  b  1
2  a  2
3  b  3

In [23]: df.groupby('A').ewm(com=1.0).mean()
Out[23]:
   B
A  
  a 0  0.000000    
  2  1.333333
  b 1  1.000000
  3  2.333333
```

Additionally, `mean` supports execution via Numba with the `engine` and `engine_kwargs` arguments. Numba must be installed as an optional dependency to use this feature.

**Other enhancements**

- Added `day_of_week` (compatibility alias `dayofweek`) property to `Timestamp, DatetimeIndex, Period, PeriodIndex` (GH9605)
- Added `day_of_year` (compatibility alias `dayofyear`) property to `Timestamp, DatetimeIndex, Period, PeriodIndex` (GH9605)
- Added `set_flags()` for setting table-wide flags on a Series or DataFrame (GH28394)
- `DataFrame.applymap()` now supports `na_action` (GH23803)
- `Index` with object dtype supports division and multiplication (GH34160)
- `io.sql.get_schema()` now supports a `schema` keyword argument that will add a schema into the create table statement (GH28486)
- `DataFrame.explode()` and `Series.explode()` now support exploding of sets (GH35614)
- `DataFrame.hist()` now supports time series (datetime) data (GH32590)
- `Styler.set_table_styles()` now allows the direct styling of rows and columns and can be chained (GH35607)
• **Styler** now allows direct CSS class name addition to individual data cells (GH36159)

• **Rolling.mean()** and **Rolling.sum()** use Kahan summation to calculate the mean to avoid numerical problems (GH10319, GH11645, GH13254, GH23761, GH36031)

• **DatetimeIndex.searchsorted()**, **TimedeltaIndex.searchsorted()**, **PeriodIndex.searchsorted()**, and **Series.searchsorted()** with datetime-like dtypes will now try to cast string arguments (list-like and scalar) to the matching datetime-like type (GH36346)

• Added methods **IntegerArray.prod()**, **IntegerArray.min()**, and **IntegerArray.max()** (GH33790)

• Calling a NumPy ufunc on a **DataFrame** with extension types now preserves the extension types when possible (GH23743)

• Calling a binary-input NumPy ufunc on multiple **DataFrame** objects now aligns, matching the behavior of binary operations and ufuncs on **Series** (GH23743). This change has been reverted in pandas 1.2.1, and the behaviour to not align DataFrames is deprecated instead, see the [1.2.1 release notes](https://pandas.pydata.org/pandas-docs/stable/releases/1.2.1.html).

• Where possible **RangeIndex.difference()** and **RangeIndex.symmetric_difference()** will return **RangeIndex** instead of **Int64Index** (GH36564)

• **DataFrame.to_parquet()** now supports **MultiIndex** for columns in parquet format (GH34777)

• **read_parquet()** gained a use_nullable_dtypes=True option to use nullable dtypes that use pd.NA as missing value indicator where possible for the resulting DataFrame (default is False, and only applicable for engine= "pyarrow") (GH31242)

• Added **Rolling.sem()** and **Expanding.sem()** to compute the standard error of the mean (GH26476)

• **Rolling.var()** and **Rolling.std()** use Kahan summation and Welford’s Method to avoid numerical issues (GH37051)

• **DataFrame.corr()** and **DataFrame.cov()** use Welford’s Method to avoid numerical issues (GH37448)

• **DataFrame.plot()** now recognizes xlabel and ylabel arguments for plots of type scatter and hexbin (GH37001)

• **DataFrame** now supports the divmod operation (GH37165)

• **DataFrame.to_parquet()** now returns a bytes object when no path argument is passed (GH37105)

• Rolling now supports the closed argument for fixed windows (GH34315)

• **DatetimeIndex** and **Series** with datetime64 or datetime64tz dtypes now support **std** (GH37436)

• Window now supports all Scipy window types in win_type with flexible keyword argument support (GH34556)

• **testing.assert_index_equal()** now has a check_order parameter that allows indexes to be checked in an order-insensitive manner (GH37478)

• **read_csv()** supports memory-mapping for compressed files (GH37621)

• Add support for min_count keyword for **DataFrame.groupby()** and **DataFrame.resample()** for functions min, max, first and last (GH37821, GH37768)

• Improve error reporting for **DataFrame.merge()** when invalid merge column definitions were given (GH16228)

• Improve numerical stability for **Rolling.skew()**, **Rolling.kurt()**, **Expanding.skew()** and **Expanding.kurt()** through implementation of Kahan summation (GH6929)

• Improved error reporting for subsetting columns of a **DataFrameGroupBy** with axis=1 (GH37725)

• Implement method cross for **DataFrame.merge()** and **DataFrame.join()** (GH5401)

• When **read_csv()**, **read_sas()** and **read_json()** are called with chunksize/iterator they can be used in a with statement as they return context-managers (GH38225)
Augmented the list of named colors available for styling Excel exports, enabling all of CSS4 colors (GH38247)

Notable bug fixes

These are bug fixes that might have notable behavior changes.

Consistency of DataFrame Reductions

`DataFrame.any()` and `DataFrame.all()` with `bool_only=True` now determines whether to exclude object-dtype columns on a column-by-column basis, instead of checking if all object-dtype columns can be considered boolean.

This prevents pathological behavior where applying the reduction on a subset of columns could result in a larger Series result. See (GH37799).

```
In [24]: df = pd.DataFrame({"A": ["foo", "bar"], "B": [True, False], dtype=object})
In [25]: df["C"] = pd.Series([True, True])
```

Previous behavior:

```
In [5]: df.all(bool_only=True)
Out[5]:
   C  True
dtype: bool

In [6]: df[["B", "C"]].all(bool_only=True)
Out[6]:
   B  False
   C  True
dtype: bool
```

New behavior:

```
In [26]: In [5]: df.all(bool_only=True)
Out[26]:
   B  False
   C  True
dtype: bool

In [27]: In [6]: df[["B", "C"]].all(bool_only=True)
Out[27]:
   B  False
   C  True
dtype: bool
```

Other DataFrame reductions with `numeric_only=None` will also avoid this pathological behavior (GH37827):

```
In [28]: df = pd.DataFrame({"A": [0, 1, 2], "B": ["a", "b", "c"], dtype=object})
```

Previous behavior:

```
In [3]: df.mean()
Out[3]: Series([], dtype: float64)
```

(continues on next page)
New behavior:

Moreover, DataFrame reductions with `numeric_only=None` will now be consistent with their Series counterparts. In particular, for reductions where the Series method raises `TypeError`, the DataFrame reduction will now consider that column non-numeric instead of casting to a NumPy array which may have different semantics (GH36076, GH28949, GH21020).

Previous behavior:

New behavior:
Increased minimum version for Python

pandas 1.2.0 supports Python 3.7.1 and higher (GH35214).

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH35214). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.16.5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pytz</td>
<td>2017.3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.7.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.6.8</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>5.0.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mypy (dev)</td>
<td>0.782</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>beautifulsoup4</td>
<td>4.6.0</td>
<td></td>
</tr>
<tr>
<td>fastparquet</td>
<td>0.3.2</td>
<td></td>
</tr>
<tr>
<td>fsspec</td>
<td>0.7.4</td>
<td></td>
</tr>
<tr>
<td>gcsfs</td>
<td>0.6.0</td>
<td></td>
</tr>
<tr>
<td>lxml</td>
<td>4.3.0</td>
<td>X</td>
</tr>
<tr>
<td>matplotlib</td>
<td>2.2.3</td>
<td>X</td>
</tr>
<tr>
<td>numba</td>
<td>0.46.0</td>
<td></td>
</tr>
<tr>
<td>openpyxl</td>
<td>2.6.0</td>
<td>X</td>
</tr>
<tr>
<td>pyarrow</td>
<td>0.15.0</td>
<td>X</td>
</tr>
<tr>
<td>pymysql</td>
<td>0.7.11</td>
<td>X</td>
</tr>
<tr>
<td>pytables</td>
<td>3.5.1</td>
<td>X</td>
</tr>
<tr>
<td>s3fs</td>
<td>0.4.0</td>
<td></td>
</tr>
<tr>
<td>scipy</td>
<td>1.2.0</td>
<td></td>
</tr>
<tr>
<td>sqlalchemy</td>
<td>1.2.8</td>
<td>X</td>
</tr>
<tr>
<td>xarray</td>
<td>0.12.3</td>
<td>X</td>
</tr>
<tr>
<td>xlrd</td>
<td>1.2.0</td>
<td>X</td>
</tr>
<tr>
<td>xlsxwriter</td>
<td>1.0.2</td>
<td>X</td>
</tr>
<tr>
<td>xlwt</td>
<td>1.3.0</td>
<td>X</td>
</tr>
<tr>
<td>pandas-gbq</td>
<td>0.12.0</td>
<td></td>
</tr>
</tbody>
</table>

See Dependencies and Optional dependencies for more.
Other API changes

- Sorting in descending order is now stable for `Series.sort_values()` and `Index.sort_values()` for Datetime-like `Index` subclasses. This will affect sort order when sorting a DataFrame on multiple columns, sorting with a key function that produces duplicates, or requesting the sorting index when using `Index.sort_values()`. When using `Series.value_counts()`, the count of missing values is no longer necessarily last in the list of duplicate counts. Instead, its position corresponds to the position in the original Series. When using `Index.sort_values()` for Datetime-like `Index` subclasses, NaTs ignored the `na_position` argument and were sorted to the beginning. Now they respect `na_position`, the default being last, same as other `Index` subclasses (GH35992)

- Passing an invalid `fill_value` to `Categorical.take()`, `DatetimeArray.take()`, `TimedeltaArray.take()`, or `PeriodArray.take()` now raises a `TypeError` instead of a `ValueError` (GH37733)

- Passing an invalid `fill_value` to `Series.shift()` with a `CategoricalDtype` now raises a `TypeError` instead of a `ValueError` (GH37733)

- Passing an invalid value to `IntervalIndex.insert()` or `CategoricalIndex.insert()` now raises a `TypeError` instead of a `ValueError` (GH37733)

- Attempting to reindex a Series with a `CategoricalIndex` with an invalid `fill_value` now raises a `TypeError` instead of a `ValueError` (GH37733)

- `CategoricalIndex.append()` with an index that contains non-category values will now cast instead of raising `TypeError` (GH38098)

Deprecations

- Deprecated parameter `inplace` in `MultiIndex.set_codes()` and `MultiIndex.set_levels()` (GH35626)

- Deprecated parameter `dtype` of method `copy()` for all `Index` subclasses. Use the `astype()` method instead for changing dtype (GH35853)

- Deprecated parameters `levels` and `codes` in `MultiIndex.copy()`. Use the `set_levels()` and `set_codes()` methods instead (GH36685)

- Date parser functions `parse_date_time()`, `parse_date_fields()`, `parse_all_fields()` and `generic_parser()` from pandas.io.date_converters are deprecated and will be removed in a future version; use `to_datetime()` instead (GH35741)

- `DataFrame.lookup()` is deprecated and will be removed in a future version, use `DataFrame.melt()` and `DataFrame.loc()` instead (GH35224)

- The method `Index.to_native_types()` is deprecated. Use `.astype(str)` instead (GH28867)

- Deprecated indexing `DataFrame` rows with a single datetime-like string as `df[string]` (given the ambiguity whether it is indexing the rows or selecting a column), use `df.loc[string]` instead (GH36179)

- Deprecated `Index.is_all_dates()` (GH27744)

- The default value of `regex` for `Series.str.replace()` will change from `True` to `False` in a future release. In addition, single character regular expressions will not be treated as literal strings when `regex=True` is set (GH24804)

- Deprecated automatic alignment on comparison operations between `DataFrame` and `Series`, do `frame, ser = frame.align(ser, axis=1, copy=False)` before e.g. `frame == ser` (GH28759)

- `Rolling.count()` with `min_periods=None` will default to the size of the window in a future version (GH31302)
• Using “outer” ufuncs on DataFrames to return 4d ndarray is now deprecated. Convert to an ndarray first (GH23743)

• Deprecated slice-indexing on tz-aware DatetimeIndex with naive datetime objects, to match scalar indexing behavior (GH36148)

• Index.ravel() returning a np.ndarray is deprecated, in the future this will return a view on the same index (GH19956)

• Deprecate use of strings denoting units with ‘M’, ‘Y’ or ‘y’ in to_timedelta() (GH36666)

• Index methods &, |, and ^ behaving as the set operations Index.intersection(), Index.union(), and Index.symmetric_difference(), respectively, are deprecated and in the future will behave as pointwise boolean operations matching Series behavior. Use the named set methods instead (GH36758)

• Categorical.is_dtype_equal() and CategoricalIndex.is_dtype_equal() are deprecated, will be removed in a future version (GH37545)

• Series.slice_shift() and DataFrame.slice_shift() are deprecated, use Series.shift() or DataFrame.shift() instead (GH37601)

• Partial slicing on unordered DatetimeIndex objects with keys that are not in the index is deprecated and will be removed in a future version (GH18531)

• The how keyword in PeriodIndex.astype() is deprecated and will be removed in a future version, use index.to_timestamp(how=how) instead (GH37982)

• Deprecated Index.asi8() for Index subclasses other than DatetimeIndex, TimedeltaIndex, and PeriodIndex (GH37877)

• The inplace parameter of Categorical.remove_unused_categories() is deprecated and will be removed in a future version (GH37643)

• The null_counts parameter of DataFrame.info() is deprecated and replaced by show_counts. It will be removed in a future version (GH37999)

Calling NumPy ufuncs on non-aligned DataFrames

Calling NumPy ufuncs on non-aligned DataFrames changed behaviour in pandas 1.2.0 (to align the inputs before calling the ufunc), but this change is reverted in pandas 1.2.1. The behaviour to not align is now deprecated instead, see the the 1.2.1 release notes for more details.

Performance improvements

• Performance improvements when creating DataFrame or Series with dtype str or StringDtype from array with many string elements (GH36304, GH36317, GH36325, GH36432, GH37371)

• Performance improvement in GroupBy.agg() with the numba engine (GH35759)

• Performance improvements when creating Series.map() from a huge dictionary (GH34717)

• Performance improvement in GroupBy.transform() with the numba engine (GH36240)

• Styler uuid method altered to compress data transmission over web whilst maintaining reasonably low table collision probability (GH36345)

• Performance improvement in to_datetime() with non-ns time unit for float dtype columns (GH20445)

• Performance improvement in setting values on an IntervalArray (GH36310)

• The internal index method _shallow_copy() now makes the new index and original index share cached attributes, avoiding creating these again, if created on either. This can speed up operations that depend on creating copies of existing indexes (GH36840)
• Performance improvement in `RollingGroupby.count()` (GH35625)
• Small performance decrease to `Rolling.min()` and `Rolling.max()` for fixed windows (GH36567)
• Reduced peak memory usage in `DataFrame.to_pickle()` when using protocol=5 in python 3.8+ (GH34244)
• Faster `dir` calls when the object has many index labels, e.g. `dir(ser)` (GH37450)
• Performance improvement in `ExpandingGroupby` (GH37064)
• Performance improvement in `Series.astype()` and `DataFrame.astype()` for `Categorical` (GH8628)
• Performance improvement in `pd.isin()` for inputs with more than 1e6 elements (GH36611)
• Performance improvement for `DataFrame.__setitem__()` with list-like indexers (GH37954)
• `read_json()` now avoids reading entire file into memory when chunksize is specified (GH34548)

**Bug fixes**

**Categorical**

• `Categorical.fillna()` will always return a copy, validate a passed fill value regardless of whether there are any NAs to fill, and disallow an `NaT` as a fill value for numeric categories (GH36530)
• Bug in `Categorical._setitem__()` that incorrectly raised when trying to set a tuple value (GH20439)
• Bug in `CategoricalIndex.equals()` incorrectly casting non-category entries to `np.nan` (GH37667)
• Bug in `CategoricalIndex.where()` incorrectly setting non-category entries to `np.nan` instead of raising `TypeError` (GH37977)
• Bug in `Categorical.to_numpy()` and `np.array(categorical)` with tz-aware datetime64 categories incorrectly dropping the time zone information instead of casting to object dtype (GH38136)

**Datetime-like**

• Bug in `DataFrame.combine_first()` that would convert datetime-like column on other `DataFrame` to integer when the column is not present in original `DataFrame` (GH28481)
• Bug in `DatetimeArray.date` where a `ValueError` would be raised with a read-only backing array (GH33530)
• Bug in `NaT` comparisons failing to raise `TypeError` on invalid inequality comparisons (GH35046)
• Bug in `DateOffset` where attributes reconstructed from pickle files differ from original objects when input values exceed normal ranges (e.g. months=12) (GH34511)
• Bug in `DatetimeIndex.get_slice_bound()` where `datetime.date` objects were not accepted or naive `Timestamp` with a tz-aware `DatetimeIndex` (GH35690)
• Bug in `DatetimeIndex.slice_locs()` where `datetime.date` objects were not accepted (GH34077)
• Bug in `DatetimeIndex.searchsorted()`, `TimedeltaIndex.searchsorted()`, `PeriodIndex.searchsorted()`, and `Series.searchsorted()` with.datetim64, timedelta64 or `Period` dtype placement of `NaT` values being inconsistent with NumPy (GH36176, GH36254)
• Inconsistency in `DatetimeArray`, `TimedeltaArray`, and `PeriodArray` method `__setitem__` casting arrays of strings to datetime-like scalars but not scalar strings (GH36261)
• Bug in DatetimeArray.take() incorrectly allowing fill_value with a mismatched time zone (GH37356)
• Bug in DatetimeIndex.shift incorrectly raising when shifting empty indexes (GH14811)
• Timestamp and DatetimeIndex comparisons between tz-aware and tz-naive objects now follow the standard
  library datetime behavior, returning True/False for !=/== and raising for inequality comparisons (GH28507)
• Bug in DatetimeIndex.equals() and TimedeltaIndex.equals() incorrectly considering int64 indexes as equal (GH36744)
• Series.to_json(), DataFrame.to_json(), and read_json() now implement time zone parsing when orient
  structure is table (GH35973)
• astype() now attempts to convert to datetime64[ns, tz] directly from object with inferred time zone
  from string (GH35973)
• Bug in TimedeltaIndex.sum() and Series.sum() with timedelta64 dtype on an empty index or series
  returning NaT instead of Timedelta(0) (GH31751)
• Bug in DatetimeArray.shift() incorrectly allowing fill_value with a mismatched time zone (GH37299)
• Bug in adding a BusinessDay with nonzero offset to a non-scalar other (GH37457)
• Bug in to_datetime() with a read-only array incorrectly raising (GH34857)
• Bug in Series.isin() with datetime64[ns] dtype and DatetimeIndex.isin() incorrectly casting integers to datetimes (GH36621)
• Bug in Series.isin() with datetime64[ns] dtype and DatetimeIndex.isin() failing to consider tz-
  aware and tz-naive datetimes as always different (GH35728)
• Bug in Series.isin() with PeriodDtype dtype and PeriodIndex.isin() failing to consider arguments
  with different PeriodDtype as always different (GH37528)
• Bug in Period constructor now correctly handles nanoseconds in the value argument (GH34621 and GH17053)

Timedelta

• Bug in TimedeltaIndex, Series, and DataFrame floor-division with timedelta64 dtypes and NaT in the
  denominator (GH35529)
• Bug in parsing of ISO 8601 durations in Timedelta and to_datetime() (GH29773, GH36204)
• Bug in to_timedelta() with a read-only array incorrectly raising (GH34857)
• Bug in Timedelta incorrectly truncating to sub-second portion of a string input when it has precision higher
  than nanoseconds (GH36738)

Timezones

• Bug in date_range() was raising AmbiguousTimeError for valid input with ambiguous=False (GH35297)
• Bug in Timestamp.replace() was losing fold information (GH37610)
Numeric

- Bug in `to_numeric()` where float precision was incorrect (GH31364)
- Bug in `DataFrame.any()` with `axis=1` and `bool_only=True` ignoring the `bool_only` keyword (GH32432)
- Bug in `Series.equals()` where a `ValueError` was raised when NumPy arrays were compared to scalars (GH35267)
- Bug in `Series` where two Series each have a `DatetimeIndex` with different time zones having those indexes incorrectly changed when performing arithmetic operations (GH33671)
- Bug in `pandas.testing` module functions when used with `check_exact=False` on complex numeric types (GH28235)
- Bug in `DataFrame.__rmatmul__()` error handling reporting transposed shapes (GH21581)
- Bug in `Series` flex arithmetic methods where the result when operating with a list, tuple or np.ndarray would have an incorrect name (GH36760)
- Bug in `IntegerArray` multiplication with `timedelta` and `np.timedelta64` objects (GH36870)
- Bug in `MultiIndex` comparison with tuple incorrectly treating tuple as array-like (GH21517)
- Bug in `DataFrame.diff()` with `datetime64` dtypes including NaT values failing to fill NaT results correctly (GH32441)
- Bug in `DataFrame` arithmetic ops incorrectly accepting keyword arguments (GH36843)
- Bug in `IntervalArray` comparisons with `Series` not returning `Series` (GH36908)
- Bug in `DataFrame` allowing arithmetic operations with list of array-likes with undefined results. Behavior changed to raising `ValueError` (GH36702)
- Bug in `DataFrame.std()` with `timedelta64` dtype and `skipna=False` (GH37392)
- Bug in `DataFrame.min()` and `DataFrame.max()` with `datetime64` dtype and `skipna=False` (GH36907)
- Bug in `DataFrame.idxmax()` and `DataFrame.idxmin()` with mixed dtypes incorrectly raising `TypeError` (GH38195)

Conversion

- Bug in `DataFrame.to_dict()` with `orient='records'` now returns python native datetime objects for datetime-like columns (GH21256)
- Bug in `Series.astype()` conversion from string to float raised in presence of `pd.NA` values (GH37626)

Strings

- Bug in `Series.to_string()`, `DataFrame.to_string()`, and `DataFrame.to_latex()` adding a leading space when `index=False` (GH24980)
- Bug in `to_numeric()` raising a `TypeError` when attempting to convert a string dtype Series containing only numeric strings and `NA` (GH37262)
Interval

- Bug in `DataFrame.replace()` and `Series.replace()` where `Interval` dtypes would be converted to object dtypes (GH34871)
- Bug in `IntervalIndex.take()` with negative indices and `fill_value=None` (GH37330)
- Bug in `IntervalIndex.putmask()` with datetime-like dtype incorrectly casting to object dtype (GH37968)
- Bug in `IntervalArray.astype()` incorrectly dropping dtype information with a `CategoricalDtype` object (GH37984)

Indexing

- Bug in `PeriodIndex.get_loc()` incorrectly raising `ValueError` on non-datelike strings instead of `KeyError`, causing similar errors in `Series.__getitem__()`, `Series.__contains__()`, and `Series.loc.__getitem__()` (GH34240)
- Bug in `Index.sort_values()` where, when empty values were passed, the method would break by trying to compare missing values instead of pushing them to the end of the sort order (GH35584)
- Bug in `Index.get_indexer()` and `Index.get_indexer_non_unique()` where `int64` arrays are returned instead of `intp` (GH36359)
- Bug in `DataFrame.sort_index()` where parameter `ascending` passed as a list on a single level index gives wrong result (GH32334)
- Bug in `DataFrame.reset_index()` was incorrectly raising a `ValueError` for input with a `MultiIndex` with missing values in a level with Categorical dtype (GH24206)
- Bug in indexing with boolean masks on datetime-like values sometimes returning a view instead of a copy (GH36210)
- Bug in `DataFrame.__getitem__()` and `DataFrame.loc.__getitem__()` with `IntervalIndex` columns and a numeric indexer (GH26490)
- Bug in `Series.__getitem__()` with a non-unique `MultiIndex` and an empty-list indexer (GH13691)
- Bug in indexing on a `Series` or `DataFrame` with a `MultiIndex` and a level named "0" (GH37194)
- Bug in `Series.__getitem__()` when using an unsigned integer array as an indexer giving incorrect results or segfaulting instead of raising `KeyError` (GH37218)
- Bug in `Index.where()` incorrectly casting numeric values to strings (GH37591)
- Bug in `DataFrame.loc()` returning empty result when indexer is a slice with negative step size (GH38071)
- Bug in `Series.loc()` and `DataFrame.loc()` raises when the index was of `object` dtype and the given numeric label was in the index (GH26491)
- Bug in `DataFrame.loc()` returned requested key plus missing values when `loc` was applied to single level from a `MultiIndex` (GH27104)
- Bug in indexing on a `Series` or `DataFrame` with a `CategoricalIndex` using a list-like indexer containing NA values (GH37722)
- Bug in `DataFrame.loc.__setitem__()` expanding an empty `DataFrame` with mixed dtypes (GH37932)
- Bug in `DataFrame.xs()` ignored `droplevel=False` for columns (GH19056)
- Bug in `DataFrame.reindex()` raising `IndexingError` wrongly for empty `DataFrame` with tolerance not `None` or method="nearest" (GH27315)
- Bug in indexing on a *Series* or *DataFrame* with a *CategoricalIndex* using list-like indexer that contains elements that are in the index's categories but not in the index itself failing to raise *KeyError* (GH37901)
- Bug on inserting a boolean label into a *DataFrame* with a numeric *Index* columns incorrectly casting to integer (GH36319)
- Bug in *DataFrame.iloc()* and *Series.iloc()* aligning objects in *__setitem__* (GH22046)
- Bug in *MultiIndex.drop()* does not raise if labels are partially found (GH37820)
- Bug in *DataFrame.loc()* did not raise *KeyError* when missing combination was given with slice(None) for remaining levels (GH19556)
- Bug in *DataFrame.loc()* raising *TypeError* when non-integer slice was given to select values from *MultiIndex* (GH25165, GH24263)
- Bug in *Series.at()* returning *Series* with one element instead of scalar when index is a *MultiIndex* with one level (GH38053)
- Bug in *DataFrame.loc()* and *Series.loc()* aligning objects in wrong order when indexer is differently ordered than the *MultiIndex* to filter (GH31330, GH34603)
- Bug in *DataFrame.loc()* and *DataFrame.__getitem__()* raising *KeyError* when columns were *MultiIndex* with only one level (GH29749)
- Bug in *Series.__getitem__()* and *DataFrame.__getitem__()* raising blank *KeyError* without missing keys for *IntervalIndex* (GH27365)
- Bug in setting a new label on a *DataFrame* or *Series* with a *CategoricalIndex* incorrectly raising *TypeError* when the new label is not among the index’s categories (GH38098)
- Bug in *Series.loc()* and *Series.iloc()* raising *ValueError* when inserting a list-like *np.array*, *list* or *tuple* in an object *Series* of equal length (GH37748, GH37486)
- Bug in *Series.loc()* and *Series.iloc()* setting all the values of an object *Series* with those of a list-like *ExtensionArray* instead of inserting it (GH38271)

**Missing**

- Bug in *SeriesGroupBy.transform()* now correctly handles missing values for dropna=False (GH35014)
- Bug in *Series.nunique()* with dropna=True was returning incorrect results when both *NA* and *None* missing values were present (GH37566)
- Bug in *Series.interpolate()* where kwarg limit_area and limit_direction had no effect when using methods pad and backfill (GH31048)

**MultiIndex**

- Bug in *DataFrame.xs()* when used with *IndexSlice* raises *TypeError* with message "Expected label or tuple of labels" (GH35301)
- Bug in *DataFrame.reset_index()* with *NaT* values in index raises *ValueError* with message "cannot convert float NaN to integer" (GH36541)
- Bug in *DataFrame.combine_first()* when used with *MultiIndex* containing string and *NaN* values raises *TypeError* (GH36562)
- Bug in *MultiIndex.drop()* dropped *NaN* values when non existing key was given as input (GH18853)
• Bug in `MultiIndex.drop()` dropping more values than expected when index has duplicates and is not sorted (GH33494)

I/O

• `read_sas()` no longer leaks resources on failure (GH35566)
• Bug in `DataFrame.to_csv()` and `Series.to_csv()` caused a `ValueError` when it was called with a filename in combination with `mode` containing a `b` (GH35058)
• Bug in `read_csv()` with `float_precision='round_trip'` did not handle decimal and thousands parameters (GH35365)
• `to_pickle()` and `read_pickle()` were closing user-provided file objects (GH35679)
• `to_csv()` passes compression arguments for 'gzip' always to `gzip.GzipFile` (GH28103)
• `to_csv()` did not support zip compression for binary file object not having a filename (GH35058)
• `to_csv()` and `read_csv()` did not honor compression and encoding for path-like objects that are internally converted to file-like objects (GH35677, GH26124, GH32392)
• `DataFrame.to_pickle()`, `Series.to_pickle()`, and `read_pickle()` did not support compression for file-objects (GH26237, GH29054, GH29570)
• Bug in `LongTableBuilder.middle_separator()` was duplicating LaTeX longtable entries in the List of Tables of a LaTeX document (GH34360)
• Bug in `read_csv()` with `engine='python'` truncating data if multiple items present in first row and first element started with BOM (GH36343)
• Removed `private_key` and `verbose` from `read_gbq()` as they are no longer supported in pandas-gbq (GH34654, GH30200)
• Bumped minimum pytables version to 3.5.1 to avoid a `ValueError` in `read_hdf()` (GH24839)
• Bug in `read_table()` and `read_csv()` when `delim_whitespace=True` and `sep=default` (GH36583)
• Bug in `DataFrame.to_json()` and `Series.to_json()` when used with `lines=True` and `orient='records'` the last line of the record is not appended with 'new line character' (GH36888)
• Bug in `read_parquet()` with fixed offset time zones. String representation of time zones was not recognized (GH35997, GH36004)
• Bug in `DataFrame.to_html()`, `DataFrame.to_string()`, and `DataFrame.to_latex()` ignoring the `na_rep` argument when `float_format` was also specified (GH9046, GH13828)
• Bug in output rendering of complex numbers showing too many trailing zeros (GH36799)
• Bug in `HDFStore` threw a `TypeError` when exporting an empty DataFrame with `datetime64[ns, tz]` dtypes with a fixed HDF5 store (GH20594)
• Bug in HDFStore was dropping time zone information when exporting a `Series` with `datetime64[ns, tz]` dtypes with a fixed HDF5 store (GH20594)
• `read_csv()` was closing user-provided binary file handles when `engine="c"` and an `encoding` was requested (GH36980)
• Bug in `DataFrame.to_hdf()` was not dropping missing rows with `dropna=True` (GH35719)
• Bug in `read_html()` was raising a `TypeError` when supplying a `pathlib.Path` argument to the `io` parameter (GH377705)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- DataFrame.to_excel(), Series.to_excel(), DataFrame.to_markdown(), and Series.to_markdown() now support writing to fsspec URLs such as S3 and Google Cloud Storage (GH33987)
- Bug in read_fwf() with skip_blank_lines=True was not skipping blank lines (GH37758)
- Parse missing values using read_json() with dtype=False to NaN instead of None (GH28501)
- read_fwf() was inferring compression with compression=None which was not consistent with the other read_* functions (GH37909)
- DataFrame.to_html() was ignoring formatters argument for ExtensionDtype columns (GH36525)
- Bumped minimum xarray version to 0.12.3 to avoid reference to the removed Panel class (GH27101, GH37983)
- DataFrame.to_csv() was re-opening file-like handles that also implement os.PathLike (GH38125)
- Bug in the conversion of a sliced pyarrow.Table with missing values to a DataFrame (GH38525)
- Bug in read_sql_table() raising a sqlalchemy.exc.OperationalError when column names contained a percentage sign (GH37517)

Period

- Bug in DataFrame.replace() and Series.replace() where Period dtypes would be converted to object dtypes (GH34871)

Plotting

- Bug in DataFrame.plot() was rotating xticklabels when subplots=True, even if the x-axis wasn’t an irregular time series (GH29460)
- Bug in DataFrame.plot() where a marker letter in the style keyword sometimes caused a ValueError (GH21003)
- Bug in DataFrame.plot.bar() and Series.plot.bar() where ticks positions were assigned by value order instead of using the actual value for numeric or a smart ordering for string (GH26186, GH11465). This fix has been reverted in pandas 1.2.1, see What's new in 1.2.1 (January 20, 2021)
- Twinned axes were losing their tick labels which should only happen to all but the last row or column of ‘externally’ shared axes (GH33819)
- Bug in Series.plot() and DataFrame.plot() was throwing a ValueError when the Series or DataFrame was indexed by a TimedeltaIndex with a fixed frequency and the x-axis lower limit was greater than the upper limit (GH37454)
- Bug in DataFrameGroupBy.boxplot() when subplots=False would raise a KeyError (GH16748)
- Bug in DataFrame.plot() and Series.plot() was overwriting matplotlib’s shared y axes behavior when no sharey parameter was passed (GH37942)
- Bug in DataFrame.plot() was raising a TypeError with ExtensionDtype columns (GH32073)
Styler

- Bug in Styler.render() HTML was generated incorrectly because of formatting error in rowspan attribute, it now matches with w3 syntax (GH38234)

Groupby/resample/rolling

- Bug in DataFrameGroupBy.count() and SeriesGroupBy.sum() returning NaN for missing categories when grouped on multiple Categoricals. Now returning 0 (GH35028)
- Bug in DataFrameGroupBy.apply() that would sometimes throw an erroneous ValueError if the grouping axis had duplicate entries (GH16646)
- Bug in DataFrame.resample() that would throw a ValueError when resampling from "D" to "24H" over a transition into daylight savings time (DST) (GH35219)
- Bug when combining methods DataFrame.groupby() with DataFrame.resample() and DataFrame.interpolate() raising a TypeError (GH35325)
- Bug in DataFrameGroupBy.apply() where a non-nuisance grouping column would be dropped from the output columns if another groupby method was called before .apply (GH34656)
- Bug when subsetting columns on a DataFrameGroupBy (e.g. df.groupby('a')['b']) would reset the attributes axis, dropna, group_keys, level, mutated, sort, and squeeze to their default values (GH9959)
- Bug in DataFrameGroupBy.tshift() failing to raise ValueError when a frequency cannot be inferred for the index of a group (GH35937)
- Bug in DataFrame.groupby() does not always maintain column index name for any, all, bfill, ffill, shift (GH29764)
- Bug in DataFrameGroupBy.apply() raising error with np.nan group(s) when dropna=False (GH35889)
- Bug in Rolling.sum() returned wrong values when dtyes where mixed between float and integer and axis=1 (GH20649, GH35596)
- Bug in Rolling.count() returned np.nan with FixedForwardWindowIndexer as window, min_periods=0 and only missing values in the window (GH35579)
- Bug where pandas.core.window.Rolling produces incorrect window sizes when using a PeriodIndex (GH34225)
- Bug in DataFrameGroupBy.ffill() and DataFrameGroupBy.bfill() where a NaN group would return filled values instead of NaN when dropna=True (GH34725)
- Bug in RollingGroupby.count() where a ValueError was raised when specifying the closed parameter (GH35869)
- Bug in DataFrameGroupBy.rolling() returning wrong values with partial centered window (GH36040)
- Bug in DataFrameGroupBy.rolling() returned wrong values with time aware window containing NaN. Raises ValueError because windows are not monotonic now (GH34617)
- Bug in Rolling.__iter__() where a ValueError was not raised when min_periods was larger than window (GH37156)
- Using Rolling.var() instead of Rolling.std() avoids numerical issues for Rolling.corr() when Rolling.var() is still within floating point precision while Rolling.std() is not (GH31286)
- Bug in DataFrameGroupBy.quantile() and Resampler.quantile() raised TypeError when values were of type Timedelta (GH29485)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in `Rolling.median()` and `Rolling.quantile()` returned wrong values for `BaseIndexer` subclasses with non-monotonic starting or ending points for windows (GH37153)
- Bug in `DataFrame.groupby()` dropped `nan` groups from result with `dropna=False` when grouping over a single column (GH35646, GH35542)
- Bug in `DataFrameGroupBy.head()`, `DataFrameGroupBy.tail()`, `SeriesGroupBy.head()`, and `SeriesGroupBy.tail()` would raise when used with `axis=1` (GH9772)
- Bug in `DataFrameGroupBy.transform()` would raise when used with `axis=1` and a transformation kernel (e.g. "shift") (GH36308)
- Bug in `DataFrameGroupBy.resample()` using `.agg` with sum produced different result than just calling `.sum` (GH35458)
- Bug in `DataFrameGroupBy.apply()` dropped values on `nan` group when returning the same axes with the original frame (GH38227)
- Bug in `DataFrameGroupBy.quantile()` couldn’t handle with arraylike `q` when grouping by columns (GH33795)
- Bug in `DataFrameGroupBy.rank()` with `datetime64tz` or period dtype incorrectly casting results to those dtypes instead of returning `float64` dtype (GH38187)

Reshaping

- Bug in `DataFrame.crosstab()` was returning incorrect results on inputs with duplicate row names, duplicate column names or duplicate names between row and column labels (GH22529)
- Bug in `DataFrame.pivot_table()` with `aggfunc='count'` or `aggfunc='sum'` returning `NaN` for missing categories when pivoted on a `Categorical`. Now returning 0 (GH31422)
- Bug in `concat()` and `DataFrame` constructor where input index names are not preserved in some cases (GH13475)
- Bug in func `crosstab()` when using multiple columns with `margins=True` and `normalize=True` (GH35144)
- Bug in `DataFrame.stack()` where an empty `DataFrame.stack` would raise an error (GH36113). Now returning an empty Series with empty MultiIndex.
- Bug in `Series.unstack()`. Now a Series with single level of Index trying to unstack would raise a `ValueError` (GH36113)
- Bug in `DataFrame.agg()` with `func={'name':<FUNC>}` incorrectly raising `TypeError` when `DataFrame.columns=['Name']` (GH36212)
- Bug in `Series.transform()` would give incorrect results or raise when the argument `func` was a dictionary (GH35811)
- Bug in `DataFrame.pivot()` did not preserve `MultiIndex` level names for columns when rows and columns are both multiindexed (GH36360)
- Bug in `DataFrame.pivot()` modified index argument when columns was passed but `values` was not (GH37635)
- Bug in `DataFrame.join()` returned a non deterministic level-order for the resulting `MultiIndex` (GH36910)
- Bug in `DataFrame.combine_first()` caused wrong alignment with dtype string and one level of MultiIndex containing only NA (GH37591)
- Fixed regression in `merge()` on merging `DatetimeIndex` with empty DataFrame (GH36895)
- Bug in `DataFrame.apply()` not setting index of return value when `func` return type is `dict` (GH37544)
• Bug in `DataFrame.merge()` and `pandas.merge()` returning inconsistent ordering in result for `how=right` and `how=left` (GH35382)

• Bug in `merge_ordered()` couldn't handle list-like `left_by` or `right_by` (GH35269)

• Bug in `merge_ordered()` returned wrong join result when length of `left_by` or `right_by` equals to the rows of `left` or `right` (GH38166)

• Bug in `merge_ordered()` didn't raise when elements in `left_by` or `right_by` not exist in `left` columns or `right` columns (GH38167)

• Bug in `DataFrame.drop_duplicates()` not validating bool dtype for `ignore_index` keyword (GH38274)

**ExtensionArray**

• Fixed bug where `DataFrame` column set to scalar extension type via a dict instantiation was considered an object type rather than the extension type (GH35965)

• Fixed bug where `astype()` with equal dtype and `copy=False` would return a new object (GH28488)

• Fixed bug when applying a NumPy ufunc with multiple outputs to an `IntegerArray` returning None (GH36913)

• Fixed an inconsistency in `PeriodArray`'s `__init__` signature to those of `DatetimeArray` and `TimedeltaArray` (GH37289)

• Reductions for `BooleanArray`, `Categorical`, `DatetimeArray`, `FloatingArray`, `IntegerArray`, `PeriodArray`, `TimedeltaArray`, and `PandasArray` are now keyword-only methods (GH37541)

• Fixed a bug where a `TypeError` was wrongly raised if a membership check was made on an `ExtensionArray` containing nan-like values (GH37867)

**Other**

• Bug in `DataFrame.replace()` and `Series.replace()` incorrectly raising an `AssertionError` instead of a `ValueError` when invalid parameter combinations are passed (GH36045)

• Bug in `DataFrame.replace()` and `Series.replace()` with numeric values and string `to_replace` (GH34789)

• Fixed metadata propagation in `Series.abs()` and ufuncs called on Series and DataFrames (GH28283)

• Bug in `DataFrame.replace()` and `Series.replace()` incorrectly casting from `PeriodDtype` to object dtype (GH34871)

• Fixed bug in metadata propagation incorrectly copying DataFrame columns as metadata when the column name overlaps with the metadata name (GH37037)

• Fixed metadata propagation in the `Series.dt`, `Series.str` accessors, `DataFrame.duplicated`, `DataFrame.stack`, `DataFrame.unstack`, `DataFrame.pivot`, `DataFrame.append`, `DataFrame.diff`, `DataFrame.applymap` and `DataFrame.update` methods (GH28283, GH37381)

• Fixed metadata propagation when selecting columns with `DataFrame.__getitem__` (GH28283)

• Bug in `Index.intersection()` with non-`Index` failing to set the correct name on the returned `Index` (GH38111)

• Bug in `RangeIndex.intersection()` failing to set the correct name on the returned `Index` in some corner cases (GH38197)

• Bug in `Index.difference()` failing to set the correct name on the returned `Index` in some corner cases (GH38268)
• Bug in `Index.union()` behaving differently depending on whether operand is an `Index` or other list-like (GH36384)
• Bug in `Index.intersection()` with non-matching numeric dtypes casting to object dtype instead of minimal common dtype (GH38122)
• Bug in `IntervalIndex.union()` returning an incorrectly-typed `Index` when empty (GH38282)
• Passing an array with 2 or more dimensions to the `Series` constructor now raises the more specific `ValueError` rather than a bare `Exception` (GH35744)
• Bug in `dir` where `dir(obj)` wouldn't show attributes defined on the instance for pandas objects (GH37173)
• Bug in `Index.drop()` raising `InvalidIndexError` when index has duplicates (GH38051)
• Bug in `RangeIndex.difference()` returning `Int64Index` in some cases where it should return `RangeIndex` (GH38028)
• Fixed bug in `assert_series_equal()` when comparing a datetime-like array with an equivalent non extension dtype array (GH37609)
• Bug in `is_bool_dtype()` would raise when passed a valid string such as "boolean" (GH38386)
• Fixed regression in logical operators raising `ValueError` when columns of `DataFrame` are a `CategoricalIndex` with unused categories (GH38367)

Contributors

A total of 257 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• 21CSM +
• AbdulMAbdi +
• Abhiraj Hinge +
• Abhishek Mangla +
• Abo7atm +
• Adam Spannbauer +
• Albert Villanova del Moral
• Alex Kirko
• Alex Lim +
• Alex Thorne +
• Aleš Erjavec +
• Ali McMaster
• Amanda Dsouza +
• Amim Knabben +
• Andrew Wieteska
• Anshoo Rajput +
• Anthony Milbourne
• Arun12121 +
- Asish Mahapatra
- Avinash Pancham +
- BeanNan +
- Ben Forbes +
- Brendan Wilby +
- Bruno Almeida +
- Byron Boulton +
- Chankey Pathak
- Chris Barnes +
- Chris Lynch +
- Chris Withers
- Christoph Deil +
- Christopher Hadley +
- Chuanzhu Xu
- Coelhudo +
- Dan Moore
- Daniel Saxton
- David Kwong +
- David Li +
- David Mrva +
- Deepak Pandey +
- Deepyaman Datta
- Devin Petersohn
- Dmitriy Perepelkin +
- Douglas Hanley +
- Dāgs Grūnbergs +
- Eli Treuherz +
- Elliot Rampono +
- Erfan Nariman
- Eric Goddard
- Eric Leung +
- Eric Wieser
- Ethan Chen +
- Eve +
- Eyal Trabelsi +
- Fabian Gebhart +
• Fangchen Li
• Felix Claessen +
• Finlay Maguire +
• Florian Roscheck +
• Gabriel Monteiro
• Gautham +
• Gerard Jorgensen +
• Gregory Livschitz
• Hans
• Harsh Sharma
• Honfung Wong +
• Igor Gotlibovych +
• Iqrar Agalosi Nureyza
• Irv Lustig
• Isaac Virshup
• Jacob Peacock
• Jacob Stevens-Haas +
• Jan Müller +
• Janus
• Jeet Parekh
• Jeff Hernandez +
• Jeff Reback
• Jiaxiang
• Joao Pedro Berno Zanutto +
• Joel Nothman
• Joel Whittier +
• John Karasinski +
• John McGuigan +
• Johnny Pribyl +
• Jonas Laursen +
• Jonathan Shreckengost +
• Joris Van den Bossche
• Jose +
• JoseNavy +
• Josh Temple +
• Jun Kudo +
• Justin Essert
• Justin Sexton +
• Kaiqi Dong
• Kamil Trocewicz +
• Karthik Mathur
• Kashif +
• Kenny Huynh
• Kevin Sheppard
• Kumar Shivam +
• Leonardus Chen +
• Levi Matus +
• Lucas Rodés-Guirao +
• Luis Pinto +
• Lynch +
• Marc Garcia
• Marco Gorelli
• Maria-Alexandra Ilie +
• Marian Denes
• Mark Graham +
• Martin Durant
• Matt Roeschke
• Matthew Roeschke
• Matthias Bussonnier
• Maxim Ivanov +
• Mayank Chaudhary +
• MeeseeksMachine
• Meghana Varanasi +
• Metehan Kutlu +
• Micael Jarniac +
• Micah Smith +
• Michael Marino
• Miroslav Šedivý
• Mohammad Jafar Mashhadi
• Mohammed Kashif +
• Nagesh Kumar C +
• Nidhi Zare +

5.3. Version 1.2
• Nikhil Choudhary +
• Number42
• Oleh Kozynets +
• OlivierLuG
• Pandas Development Team
• Paolo Lammens +
• Paul Ganssle
• Pax +
• Peter Liu +
• Philip Cerles +
• Pranjal Bhardwaj +
• Prayag Savsani +
• Purushothaman Srikanth +
• Qbiwan +
• Rahul Chauhan +
• Rahul Sathanapalli +
• Rajat Bishnoi +
• Ray Bell
• Reshama Shaikh +
• Richard Shadrach
• Robert Bradshaw
• Robert de Vries
• Rohith295
• S Mono +
• S.TAKENO +
• Sahid Velji +
• Sam Cohen +
• Sam Ezebunandu +
• Sander +
• Sarthak +
• Sarthak Vineet Kumar +
• Satrio H Wicaksono +
• Scott Lasley
• Shao Yang Hong +
• Sharon Woo +
• Shubham Mehra +
• Simon Hawkins
• Sixuan (Cherie) Wu +
• Souris Ash +
• Steffen Rehberg
• Suvayu Ali
• Sven
• SylvainLan +
• T. JEGHAM +
• Terji Petersen
• Thomas Dickson +
• Thomas Heavey +
• Thomas Smith
• Tobias Pitters
• Tom Augspurger
• Tomasz Sakrejda +
• Torsten Wörtwein +
• Ty Mick +
• UrielMaD +
• Uwe L. Korn
• Vikramaditya Gaonkar +
• VirosaLi +
• W.R +
• Warren White +
• Wesley Boelrijk +
• William Ayd
• Yanxian Lin +
• Yassir Karroum +
• Yong Kai Yi +
• Yuanhao Geng +
• Yury Mikhailov +
• Yutaro Ikeda
• Yuya Takashina +
• Zach Brookler +
• Zak Kohler +
• ZhuhuiChen0903 +
• abmyii

5.3. Version 1.2
pandas: powerful Python data analysis toolkit, Release 1.4.2

- alextn +
- asharma13524 +
- attack68
- beanan +
- chinhwee
- cleconte987
- danchev +
- ebardie +
- edwardkong
- elliot rampono +
- estasney +
- gabicca
- geetha-rangaswamaiah +
- gfyoun
- guru kiran
- hardikpns
- icanhazcodeplz +
- ivanovmg +
- jbrockmendel
- jeschwar
- jnecus
- joooeey +
- junk +
- krajatcl +
- lacrosse91 +
- leo +
- lpkirwin +
- lrjball
- lucasrodes +
- ma3da +
- mavismonica +
- mlondschen +
- mzeitlin11 +
- nguevara +
- nreben
- parkdj1 +
5.4 Version 1.1

5.4.1 What’s new in 1.1.5 (December 07, 2020)

These are the changes in pandas 1.1.5. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in addition of a timedelta-like scalar to a DatetimeIndex raising incorrectly (GH37295)
- Fixed regression in Series.groupby() raising when the Index of the Series had a tuple as its name (GH37755)
- Fixed regression in DataFrame.loc() and Series.loc() for __setitem__ when one-dimensional tuple was given to select from MultiIndex (GH37711)
- Fixed regression in inplace operations on Series with ExtensionDtype with NumPy dtyped operand (GH37910)
pandas: powerful Python data analysis toolkit, Release 1.4.2

• Fixed regression in metadata propagation for groupby iterator (GH37343)
• Fixed regression in MultiIndex constructed from a DatetimeIndex not retaining frequency (GH35563)
• Fixed regression in Index constructor raising an AttributeError when passed a SparseArray with datetime64 values (GH35843)
• Fixed regression in DataFrame.unstack() with columns with integer dtype (GH37115)
• Fixed regression in indexing on a Series with CategoricalDtype after unpickling (GH37631)
• Fixed regression in DataFrame.groupby() aggregation with out-of-bounds datetime objects in an object-dtype column (GH36003)
• Fixed regression in df.groupby(..).rolling(..) with the resulting MultiIndex when grouping by a label that is in the index (GH37641)
• Fixed regression in DataFrame.fillna() not filling NaN after other operations such as DataFrame.pivot() (GH36495).
• Fixed performance regression in df.groupby(..).rolling(..) (GH38038)
• Fixed regression in MultiIndex.intersection() returning duplicates when at least one of the indexes had duplicates (GH36915)
• Fixed regression in GroupBy.first() and GroupBy.last() where None was considered a non-NA value (GH38286)

Bug fixes

• Bug in pytables methods in python 3.9 (GH38041)

Other

• Only set -Werror as a compiler flag in the CI jobs (GH33315, GH33314)

Contributors

A total of 12 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Andrew Wieteska
• Fangchen Li
• Janus
• Joris Van den Bossche
• Matthew Roeschke
• MeeseeksMachine
• Pandas Development Team
• Richard Shadrach
• Simon Hawkins
• Uwe L. Korn
• jbrockmendel
5.4.2 What's new in 1.1.4 (October 30, 2020)

These are the changes in pandas 1.1.4. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `read_csv()` raising a `ValueError` when names was of type `dict_keys` (GH36928)
- Fixed regression in `read_csv()` with more than 1M rows and specifying a `index_col` argument (GH37094)
- Fixed regression where attempting to mutate a `DateOffset` object would no longer raise an `AttributeError` (GH36940)
- Fixed regression where `DataFrame.agg()` would fail with `TypeError` when passed positional arguments to be passed on to the aggregation function (GH36948).
- Fixed regression in `RollingGroupby` with `sort=False` not being respected (GH36889)
- Fixed regression in `Series.astype()` converting `None` to "nan" when casting to string (GH36904)
- Fixed regression in `Series.rank()` method failing for read-only data (GH37290)
- Fixed regression in `RollingGroupby` causing a segmentation fault with Index of dtype object (GH36727)
- Fixed regression in `DataFrame.resample(...) .apply(...)()` raised `AttributeError` when input was a `DataFrame` and only a `Series` was evaluated (GH36951)
- Fixed regression in `DataFrame.groupby(...) .std()` with nullable integer dtype (GH37415)
- Fixed regression in `PeriodDtype` comparing both equal and unequal to its string representation (GH37265)
- Fixed regression where slicing `DatetimeIndex` raised `AssertionError` on irregular time series with `pd.NaT` or on unsorted indices (GH36953 and GH35509)
- Fixed regression in certain offsets (`pd.offsets.Day()` and below) no longer being hashable (GH37267)
- Fixed regression in `StataReader` which required `chunksize` to be manually set when using an iterator to read a dataset (GH37280)
- Fixed regression in `setitem` with `DataFrame.iloc()` which raised error when trying to set a value while filtering with a boolean list (GH36741)
- Fixed regression in `setitem` with a Series getting aligned before setting the values (GH37427)
- Fixed regression in `MultiIndex.is_monotonic_increasing` returning wrong results with `NaN` in at least one of the levels (GH37220)
- Fixed regression in `inplace` arithmetic operation on a Series not updating the parent DataFrame (GH36373)
Bug fixes

- Bug causing groupby(...).sum() and similar to not preserve metadata (GH29442)
- Bug in Series.isin() and DataFrame.isin() raising a ValueError when the target was read-only (GH37174)
- Bug in GroupBy.fillna() that introduced a performance regression after 1.0.5 (GH36757)
- Bug in DataFrame.info() was raising a KeyError when the DataFrame has integer column names (GH37245)
- Bug in DataFrameGroupby.apply() would drop a CategoricalIndex when grouped on (GH35792)

Contributors

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Daniel Saxton
- Fangchen Li
- Janus +
- Joris Van den Bossche
- Kevin Sheppard
- Marco Gorelli
- Matt Roeschke
- Matthew Roeschke
- MeeseeksMachine
- Pandas Development Team
- Paul Ganssle
- Richard Shadrach
- Simon Hawkins
- Thomas Smith
- Tobias Pitters
- abmyii +
- jbrockmendel
- patrick
5.4.3 What’s new in 1.1.3 (October 5, 2020)

These are the changes in pandas 1.1.3. See Release notes for a full changelog including other versions of pandas.

Enhancements

Added support for new Python version

pandas 1.1.3 now supports Python 3.9 (GH36296).

Development Changes

- The minimum version of Cython is now the most recent bug-fix version (0.29.21) (GH36296).

Fixed regressions

- Fixed regression in `DataFrame.agg()`, `DataFrame.apply()`, `Series.agg()`, and `Series.apply()` where internal suffix is exposed to the users when no relabelling is applied (GH36189)
- Fixed regression in `IntegerArray` unary plus and minus operations raising a TypeError (GH36063)
- Fixed regression when adding a `timedelta_range()` to a `Timestamp` raised a ValueError (GH35897)
- Fixed regression in `Series.__getitem__()` incorrectly raising when the input was a tuple (GH35534)
- Fixed regression in `Series.__getitem__()` incorrectly raising when the input was a frozenset (GH35747)
- Fixed regression in modulo of `Index`, `Series` and `DataFrame` using numexpr using C not Python semantics (GH36047, GH36526)
- Fixed regression in `read_excel()` with engine="odf" caused UnboundLocalError in some cases where cells had nested child nodes (GH36122, GH35802)
- Fixed regression in `DataFrame.replace()` inconsistent replace when using a float in the replace method (GH35376)
- Fixed regression in `Series.loc()` on a `Series` with a `MultiIndex` containing `Timestamp` raising `InvalidIndexError` (GH35858)
- Fixed regression in `DataFrame` and `Series` comparisons between numeric arrays and strings (GH35700, GH36377)
- Fixed regression in `DataFrame.apply()` with raw=True and user-function returning string (GH35940)
- Fixed regression when setting empty `DataFrame` column to a `Series` in preserving name of index in frame (GH36527)
- Fixed regression in `Period` incorrect value for ordinal over the maximum timestamp (GH36430)
- Fixed regression in `read_table()` raised `ValueError` when delim_whitespace was set to True (GH35958)
- Fixed regression in `Series.dt.normalize()` when normalizing pre-epoch dates the result was shifted one day (GH36294)
Bug fixes

- Bug in `read_spss()` where passing a `pathlib.Path` as `path` would raise a `TypeError` (GH33666)
- Bug in `Series.str.startswith()` and `Series.str.endswith()` with `category` dtype not propagating `na` parameter (GH36241)
- Bug in `Series` constructor where integer overflow would occur for sufficiently large scalar inputs when an index was provided (GH36291)
- Bug in `DataFrame.sort_values()` raising an `AttributeError` when sorting on a key that casts column to categorical dtype (GH36383)
- Bug in `DataFrame.stack()` raising a `ValueError` when stacking `MultiIndex` columns based on position when the levels had duplicate names (GH36353)
- Bug in `Series.astype()` showing too much precision when casting from `np.float32` to string dtype (GH36451)
- Bug in `Series.isin()` and `DataFrame.isin()` when using `NaN` and a row length above 1,000,000 (GH22205)
- Bug in `cut()` raising a `ValueError` when passed a `Series` of labels with `ordered=False` (GH36603)

Other

- Reverted enhancement added in pandas-1.1.0 where `timedelta_range()` infers a frequency when passed `start`, `stop`, and `periods` (GH32377)

Contributors

A total of 16 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Asish Mahapatra
- Dan Moore +
- Daniel Saxton
- Fangchen Li
- Hans
- Irv Lustig
- Joris Van den Bossche
- Kaiqi Dong
- MeeseeksMachine
- Number42 +
- Pandas Development Team
- Richard Shadrach
- Simon Hawkins
- jbrockmendel
- nrebena
- patrick
5.4.4 What’s new in 1.1.2 (September 8, 2020)

These are the changes in pandas 1.1.2. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Regression in `DatetimeIndex.intersection()` incorrectly raising `AssertionError` when intersecting against a list (GH35876)
- Fix regression in updating a column inplace (e.g. using `df['col'].fillna(..., inplace=True)`) (GH35731)
- Fix regression in `DataFrame.append()` mixing tz-aware and tz-naive datetime columns (GH35460)
- Performance regression for `RangeIndex.format()` (GH35712)
- Regression where `MultiIndex.get_loc()` would return a slice spanning the full index when passed an empty list (GH35878)
- Fix regression in invalid cache after an indexing operation; this can manifest when setting which does not update the data (GH35521)
- Regression in `DataFrame.replace()` where a `TypeError` would be raised when attempting to replace elements of type `Interval` (GH35931)
- Fix regression in pickle roundtrip of the closed attribute of `IntervalIndex` (GH35658)
- Fixed regression in `DataFrameGroupBy.agg()` where a `ValueError: buffer source array is read-only` would be raised when the underlying array is read-only (GH36014)
- Fixed regression in `Series.groupby.rolling()` number of levels of `MultiIndex` in input was compressed to one (GH36018)
- Fixed regression in `DataFrameGroupBy` on an empty `DataFrame` (GH36197)

Bug fixes

- Bug in `DataFrame.eval()` with object dtype column binary operations (GH35794)
- Bug in `Series` constructor raising a `TypeError` when constructing sparse datetime64 dtypes (GH35762)
- Bug in `DataFrame.apply()` with `result_type="reduce"` returning with incorrect index (GH35683)
- Bug in `Series.astype()` and `DataFrame.astype()` not respecting the `errors` argument when set to "ignore" for extension dtypes (GH35471)
- Bug in `DateTimeIndex.format()` and `PeriodIndex.format()` with `name=True` setting the first item to "None" where it should be "" (GH35712)
- Bug in `Float64Index.__contains__()` incorrectly raising `TypeError` instead of returning `False` (GH35788)
- Bug in `Series` constructor incorrectly raising a `TypeError` when passed an ordered set (GH36044)
- Bug in `Series.dt.isocalendar()` and `DatetimeIndex.isocalendar()` that returned incorrect year for certain dates (GH36032)
- Bug in `DataFrame` indexing returning an incorrect `Series` in some cases when the series has been altered and a cache not invalidated (GH33675)
- Bug in `DataFrame.corr()` causing subsequent indexing lookups to be incorrect (GH35882)
• Bug in `import_optional_dependency()` returning incorrect package names in cases where package name is different from import name (GH35948)
• Bug when setting empty `DataFrame` column to a `Series` in preserving name of index in frame (GH31368)

Other

• `factorize()` now supports `na_sentinel=None` to include NaN in the uniques of the values and remove `dropna` keyword which was unintentionally exposed to public facing API in 1.1 version from `factorize()` (GH35667)
• `DataFrame.plot()` and `Series.plot()` raise `UserWarning` about usage of `FixedFormatter` and `FixedLocator` (GH35684 and GH35945)

Contributors

A total of 16 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Ali McMaster
• Asish Mahapatra
• Daniel Saxton
• Fangchen Li
• Harsh Sharma +
• Irv Lustig
• Jeet Parekh +
• Joris Van den Bossche
• Kaiqi Dong
• Matthew Roeschke
• MeeseeksMachine
• Pandas Development Team
• Simon Hawkins
• Terji Petersen
• jbrockmendel
• patrick
5.4.5 What’s new in 1.1.1 (August 20, 2020)

These are the changes in pandas 1.1.1. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `CategoricalIndex.format()` where, when stringified scalars had different lengths, the shorter string would be right-filled with spaces, so it had the same length as the longest string (GH35439)
- Fixed regression in `Series.truncate()` when trying to truncate a single-element series (GH35544)
- Fixed regression where `DataFrame.to_numpy()` would raise a `RuntimeError` for mixed dtypes when converting to `str` (GH35455)
- Fixed regression where `read_csv()` would raise a `ValueError` when `pandas.options.mode.use_inf_as_na` was set to `True` (GH35493)
- Fixed regression where `pandas.testing.assert_series_equal()` would raise an error when non-numeric dtypes were passed with `check_exact=True` (GH35446)
- Fixed regression in `.groupby(..).rolling(..)` where column selection was ignored (GH35486)
- Fixed regression where `DataFrame.interpolate()` would raise a `TypeError` when the `DataFrame` was empty (GH35598)
- Fixed regression in `DataFrame.shift()` with `axis=1` and heterogeneous dtypes (GH35488)
- Fixed regression in `DataFrame.diff()` with read-only data (GH35559)
- Fixed regression in `.groupby(..).rolling(..)` where a segfault would occur with `center=True` and an odd number of values (GH35552)
- Fixed regression in `DataFrame.apply()` where functions that altered the input in-place only operated on a single row (GH35462)
- Fixed regression in `DataFrame.reset_index()` would raise a `ValueError` on empty `DataFrame` with a `MultiIndex` with a `datetime64` dtype level (GH35606, GH35657)
- Fixed regression where `pandas.merge_asof()` would raise a `UnboundLocalError` when `left_index`, `right_index` and `tolerance` were set (GH35558)
- Fixed regression in `.groupby(..).rolling(..)` where a custom `BaseIndexer` would be ignored (GH35557)
- Fixed regression in `DataFrame.replace()` and `Series.replace()` where compiled regular expressions would be ignored during replacement (GH35680)
- Fixed regression in `aggregate()` where a list of functions would produce the wrong results if at least one of the functions did not aggregate (GH35490)
- Fixed memory usage issue when instantiating large `pandas.arrays.StringArray` (GH35499)
Bug fixes

- Bug in *Styler* whereby `cell_ids` argument had no effect due to other recent changes (GH35588) (GH35663)
- Bug in `pandas.testing.assert_series_equal()` and `pandas.testing.assert_frame_equal()` where extension dtypes were not ignored when `check_dtypes` was set to `False` (GH35715)
- Bug in `to_timedelta()` fails when `arg` is a `Series` with `Int64` dtype containing null values (GH35574)
- Bug in `.groupby(..).rolling(..)` where passing `closed` with column selection would raise a `ValueError` (GH35549)
- Bug in `DataFrame` constructor failing to raise `ValueError` in some cases when `data` and `index` have mismatched lengths (GH33437)

Contributors

A total of 20 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Ali McMaster
- Daniel Saxton
- Eric Goddard +
- Fangchen Li
- Isaac Virshup
- Joris Van den Bossche
- Kevin Sheppard
- Matthew Roeschke
- MeeseeksMachine +
- Pandas Development Team
- Richard Shadrach
- Simon Hawkins
- Terji Petersen
- Tom Augspurger
- Yutaro Ikeda +
- attack68 +
- edwardkong +
- gabicca +
- jbrockmendel
- sanderland +
5.4.6 What’s new in 1.1.0 (July 28, 2020)

These are the changes in pandas 1.1.0. See Release notes for a full changelog including other versions of pandas.

Enhancements

**KeyErrors raised by `loc` specify missing labels**

Previously, if labels were missing for a `.loc` call, a KeyError was raised stating that this was no longer supported. Now the error message also includes a list of the missing labels (max 10 items, display width 80 characters). See GH34272.

**All dtypes can now be converted to `StringDtype`**

Previously, declaring or converting to `StringDtype` was in general only possible if the data was already only `str` or `nan-like` (GH31204). `StringDtype` now works in all situations where `astype(str)` or `dtype=str` work:

For example, the below now works:

```python
In [1]: ser = pd.Series([1, "abc", np.nan], dtype="string")
In [2]: ser
Out[2]:
0   1
1   abc
2   <NA>
dtype: string
In [3]: ser[0]
Out[3]: '1'
In [4]: pd.Series([1, 2, np.nan], dtype="Int64").astype("string")
Out[4]:
0   1
1   2
2   <NA>
dtype: string
```

**Non-monotonic PeriodIndex partial string slicing**

`PeriodIndex` now supports partial string slicing for non-monotonic indexes, mirroring DatetimeIndex behavior (GH31096)

For example:

```python
In [5]: dti = pd.date_range("2014-01-01", periods=30, freq="30D")
In [6]: pi = dti.to_period("D")
In [7]: ser_monotonic = pd.Series(np.arange(30), index=pi)
```

(continues on next page)
In [8]: shuffler = list(range(0, 30, 2)) + list(range(1, 31, 2))

In [9]: ser = ser_monotonic[shuffler]

In [10]: ser
Out[10]:
2014-01-01  0
2014-03-02  2
2014-05-01  4
2014-06-30  6
2014-08-29  8
   ..
2015-09-23 21
2015-11-22 23
2016-01-21 25
2016-03-21 27
2016-05-20 29
Freq: D, Length: 30, dtype: int64

In [11]: ser["2014"]
Out[11]:
2014-01-01  0
2014-03-02  2
2014-05-01  4
2014-06-30  6
2014-08-29  8
2014-10-28 10
2014-12-27 12
2014-01-31  1
2014-04-01  3
2014-05-31  5
2014-07-30  7
2014-09-28  9
2014-11-27 11
Freq: D, dtype: int64

In [12]: ser.loc["May 2015"]
Out[12]:
2015-05-26  17
Freq: D, dtype: int64
Comparing two DataFrame or two Series and summarizing the differences

We’ve added DataFrame.compare() and Series.compare() for comparing two DataFrame or two Series (GH30429)

```python
In [13]: df = pd.DataFrame(
        ....:     {   
        ....:         "col1": ["a", "a", "b", "b", "a"],
        ....:         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
        ....:         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
        ....:     },
        ....:     columns=["col1", "col2", "col3"],
        ....: )

In [14]: df
Out[14]:
          col1 col2  col3
0         a   1.0  1.0
1         a   2.0  2.0
2         b   3.0  3.0
3         b    NaN  4.0
4         a   5.0  5.0

In [15]: df2 = df.copy()

In [16]: df2.loc[0, 'col1'] = 'c'

In [17]: df2.loc[2, 'col3'] = 4.0

In [18]: df2
Out[18]:
          col1 col2  col3
0         c   1.0  1.0
1         a   2.0  2.0
2         b   3.0  4.0
3         b    NaN  4.0
4         a   5.0  5.0

In [19]: df.compare(df2)
Out[19]:
          col1 col3
self other self other
0  a  c  NaN  NaN
2  NaN NaN  3.0  4.0

See User Guide for more details.

5.4. Version 1.1
Allow NA in groupby key

With `groupby`, we've added a `dropna` keyword to `DataFrame.groupby()` and `Series.groupby()` in order to allow NA values in group keys. Users can define `dropna` to `False` if they want to include NA values in groupby keys. The default is set to `True` for `dropna` to keep backwards compatibility (GH3729).

```python
In [20]: df_list = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
In [21]: df_dropna = pd.DataFrame(df_list, columns=['a', 'b', 'c'])
In [22]: df_dropna
Out[22]:
      a   b   c
0    1  2.0  3
1    1  NaN  4
2    2  1.0  3
3    1  2.0  2

# Default `dropna` is set to True, which will exclude NaNs in keys
In [23]: df_dropna.groupby(by=['b'], dropna=True).sum()
Out[23]:
   a  c
b  
  1.0  2  3
  2.0  2  5

# In order to allow NaN in keys, set `dropna` to False
In [24]: df_dropna.groupby(by=['b'], dropna=False).sum()
Out[24]:
   a  c
b  
  1.0  2  3
  2.0  2  5
  NaN  1  4
```

The default setting of `dropna` argument is `True` which means NA are not included in group keys.

Sorting with keys

We've added a key argument to the `DataFrame` and `Series` sorting methods, including `DataFrame.sort_values()`, `DataFrame.sort_index()`, `Series.sort_values()`, and `Series.sort_index()`. The key can be any callable function which is applied column-by-column to each column used for sorting, before sorting is performed (GH27237). See `sort_values with keys` and `sort_index with keys` for more information.

```python
In [25]: s = pd.Series(['C', 'a', 'B'])
In [26]: s
Out[26]:
0  C
1  a
2  B
dtype: object
```

2990 Chapter 5. Release notes
In [27]: s.sort_values()
Out[27]:
2   B
0   C
1   a
dtype: object

Note how this is sorted with capital letters first. If we apply the `Series.str.lower()` method, we get

In [28]: s.sort_values(key=lambda x: x.str.lower())
Out[28]:
1   a
2   B
0   C
dtype: object

When applied to a `DataFrame`, they key is applied per-column to all columns or a subset if `by` is specified, e.g.

In [29]: df = pd.DataFrame({'a': ['C', 'C', 'a', 'a', 'B', 'B'],
                           'b': [1, 2, 3, 4, 5, 6]})
In [30]: df
Out[30]:
   a  b
0  C  1
1  C  2
2  a  3
3  a  4
4  B  5
5  B  6

In [31]: df.sort_values(by=['a'], key=lambda col: col.str.lower())
Out[31]:
   a  b
0  C  1
1  C  2
2  a  3
3  a  4
4  B  5
5  B  6
0  C  1
1  C  2

For more details, see examples and documentation in `DataFrame.sort_values()`, `Series.sort_values()`, and `sort_index()`.
Fold argument support in Timestamp constructor

Timestamp now supports the keyword-only fold argument according to PEP 495 similar to parent datetime.datetime class. It supports both accepting fold as an initialization argument and inferring fold from other constructor arguments (GH25057, GH31338). Support is limited to dateutil timezones as pytz doesn’t support fold.

For example:

```python
In [32]: ts = pd.Timestamp("2019-10-27 01:30:00+00:00")
In [33]: ts.fold
Out[33]: 0
In [34]: ts = pd.Timestamp(year=2019, month=10, day=27, hour=1, minute=30, tz="dateutil/Europe/London", fold=1)
In [35]: ts
Out[35]: Timestamp('2019-10-27 01:30:00+0000', tz='dateutil//usr/share/zoneinfo/Europe/˓London')
```

For more on working with fold, see Fold subsection in the user guide.

Parsing timezone-aware format with different timezones in to_datetime

to_datetime() now supports parsing formats containing timezone names (%Z) and UTC offsets (%z) from different timezones then converting them to UTC by setting utc=True. This would return a DatetimeIndex with timezone at UTC as opposed to an Index with object dtype if utc=True is not set (GH32792).

For example:

```python
In [36]: tz_strs = ["2010-01-01 12:00:00 +0100", "2010-01-01 12:00:00 -0100", "2010-01-01 12:00:00 +0300", "2010-01-01 12:00:00 +0400"]
In [37]: pd.to_datetime(tz_strs, format='%Y-%m-%d %H:%M:%S %z', utc=True)
Out[37]: DatetimeIndex(['2010-01-01 11:00:00+00:00', '2010-01-01 13:00:00+00:00', '2010-01-01 09:00:00+00:00', '2010-01-01 08:00:00+00:00'], dtype='datetime64[ns, UTC]', freq=None)
In [38]: pd.to_datetime(tz_strs, format='%Y-%m-%d %H:%M:%S %z')
Out[38]: Index([2010-01-01 12:00:00+00:00, 2010-01-01 12:00:00-01:00, 2010-01-01 12:00:00+03:00, 2010-01-01 12:00:00+04:00], dtype='object')
```
Grouper and resample now supports the arguments origin and offset

Grouper and DataFrame.resample() now supports the arguments origin and offset. It let the user control the timestamp on which to adjust the grouping. (GH31809)

The bins of the grouping are adjusted based on the beginning of the day of the time series starting point. This works well with frequencies that are multiples of a day (like 30D) or that divides a day (like 90s or 1min). But it can create inconsistencies with some frequencies that do not meet this criteria. To change this behavior you can now specify a fixed timestamp with the argument origin.

Two arguments are now deprecated (more information in the documentation of DataFrame.resample()):

- base should be replaced by offset.
- loffset should be replaced by directly adding an offset to the index DataFrame after being resampled.

Small example of the use of origin:

```
In [39]: start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'

In [40]: middle = '2000-10-02 00:00:00'

In [41]: rng = pd.date_range(start, end, freq='7min')

In [42]: ts = pd.Series(np.arange(len(rng)) * 3, index=rng)

In [43]: ts
Out[43]:
2000-10-01 23:30:00    0
2000-10-01 23:37:00    3
2000-10-01 23:44:00    6
2000-10-01 23:51:00    9
2000-10-01 23:58:00   12
2000-10-02 00:05:00   15
2000-10-02 00:12:00   18
2000-10-02 00:19:00   21
2000-10-02 00:26:00   24
Freq: 7T, dtype: int64
```

Resample with the default behavior 'start_day' (origin is 2000-10-01 00:00:00):

```
In [44]: ts.resample('17min').sum()
Out[44]:
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
2000-10-02 00:22:00   24
Freq: 17T, dtype: int64
```

```
In [45]: ts.resample('17min', origin='start_day').sum()
Out[45]:
2000-10-01 23:14:00    0
2000-10-01 23:31:00    9
2000-10-01 23:48:00   21
2000-10-02 00:05:00   54
```

(continues on next page)
Resample using a fixed origin:

```
In [46]: ts.resample('17min', origin='epoch').sum()
Out[46]:
2000-10-01 23:18:00    0
2000-10-01 23:35:00   18
2000-10-01 23:52:00   27
2000-10-02 00:09:00   39
2000-10-02 00:26:00   24
Freq: 17T, dtype: int64
```

```
In [47]: ts.resample('17min', origin='2000-01-01').sum()
Out[47]:
2000-10-01 23:24:00    3
2000-10-01 23:41:00   15
2000-10-01 23:58:00   45
2000-10-02 00:15:00   45
Freq: 17T, dtype: int64
```

If needed you can adjust the bins with the argument `offset` (a `Timedelta`) that would be added to the default `origin`.

For a full example, see: *Use origin or offset to adjust the start of the bins.*

### fsspec now used for filesystem handling

For reading and writing to filesystems other than local and reading from HTTP(S), the optional dependency `fsspec` will be used to dispatch operations (GH33452). This will give unchanged functionality for S3 and GCS storage, which were already supported, but also add support for several other storage implementations such as Azure Datalake and Blob, SSH, FTP, dropbox and github. For docs and capabilities, see the `fsspec` docs.

The existing capability to interface with S3 and GCS will be unaffected by this change, as `fsspec` will still bring in the same packages as before.

### Other enhancements

- Compatibility with matplotlib 3.3.0 (GH34850)
- `IntegerArray.astype()` now supports `datetime64` dtype (GH32538)
- `IntegerArray` now implements the `sum` operation (GH33172)
- Added `pandas.errors.InvalidIndexError` (GH34570).
- Added `DataFrame.value_counts()` (GH5377)
- Added a `pandas.api.indexers.FixedForwardWindowIndexer()` class to support forward-looking windows during rolling operations.
- Added a `pandas.api.indexers.VariableOffsetWindowIndexer()` class to support rolling operations with non-fixed offsets (GH34994)
- **describe()** now includes a `datetime_is_numeric` keyword to control how datetime columns are summarized (GH30164, GH34798)
- **Styler** may now render CSS more efficiently where multiple cells have the same styling (GH30876)
- **highlight_null()** now accepts `subset` argument (GH31345)
- When writing directly to a sqlite connection `DataFrame.to_sql()` now supports the `multi` method (GH29921)
- **pandas.errors.OptionError** is now exposed in `pandas.errors` (GH27553)
- Added `api.extensions.ExtensionArray.argmax()` and `api.extensions.ExtensionArray.argmin()` (GH24382)
- `timedelta_range()` will now infer a frequency when passed `start`, `stop`, and `periods` (GH32377)
- Positional slicing on a `IntervalIndex` now supports slices with `step > 1` (GH31658)
- **Series.str** now has a `fullmatch` method that matches a regular expression against the entire string in each row of the `Series`, similar to `re.fullmatch` (GH32806).
- `DataFrame.sample()` will now also allow array-like and BitGenerator objects to be passed to `random_state` as seeds (GH32503)
- `Index.union()` will now raise `RuntimeWarning` for `MultiIndex` objects if the object inside are unsortable. Pass `sort=False` to suppress this warning (GH33015)
- Added `Series.dt.isocalendar()` and `DatetimeIndex.isocalendar()` that returns a `DataFrame` with `year`, `week`, and `day` calculated according to the ISO 8601 calendar (GH33206, GH34392).
- The `DataFrame.to_feather()` method now supports additional keyword arguments (e.g. to set the compression) that are added in pyarrow 0.17 (GH33422).
- The `cut()` will now accept parameter `ordered` with default `ordered=True`. If `ordered=False` and no labels are provided, an error will be raised (GH33141)
- `DataFrame.to_csv()`, `DataFrame.to_pickle()`, and `DataFrame.to_json()` now support passing a dict of compression arguments when using the gzip and bz2 protocols. This can be used to set a custom compression level, e.g., `df.to_csv(path, compression={'method': 'gzip', 'compresslevel': 1})` (GH33196)
- `melt()` has gained an `ignore_index` (default True) argument that, if set to False, prevents the method from dropping the index (GH17440).
- `Series.update()` now accepts objects that can be coerced to a `Series`, such as dict and list, mirroring the behavior of `DataFrame.update()` (GH33215)
- `transform()` and `aggregate()` have gained `engine` and `engine_kwargs` arguments that support executing functions with Numba (GH32854, GH33388)
- `interpolate()` now supports SciPy interpolation method `scipy.interpolate.CubicSpline` as method `cubicspline` (GH33670)
- `DataFrameGroupBy` and `SeriesGroupBy` now implement the `sample` method for doing random sampling within groups (GH31775)
- `DataFrame.to_numpy()` now supports the `na_value` keyword to control the NA sentinel in the output array (GH33820)
- Added `api.extension.ExtensionArray.equals` to the extension array interface, similar to `Series.equals()` (GH27081)
- The minimum supported dta version has increased to 105 in `read_stata()` and `StataReader` (GH26667).
• to_stata() supports compression using the compression keyword argument. Compression can either be inferred or explicitly set using a string or a dictionary containing both the method and any additional arguments that are passed to the compression library. Compression was also added to the low-level Stata-file writers StataWriter, StataWriter117, and StataWriterUTF8 (GH26599).

• HDFStore.put() now accepts a track_times parameter. This parameter is passed to the create_table method of PyTables (GH32682).

• Series.plot() and DataFrame.plot() now accept xlabel and ylabel parameters to present labels on x and y axis (GH9093).

• Made pandas.core.window.rolling.Rolling and pandas.core.window.expanding.Expanding iterable (GH11704)

• Made option_context a contextlib.ContextDecorator, which allows it to be used as a decorator over an entire function (GH34253).

• DataFrame.to_csv() and Series.to_csv() now accept an errors argument (GH22610)

• transform() now allows func to be pad, backfill and cumcount (GH31269).

• read_json() now accepts an nrows parameter. (GH33916).

• DataFrame.hist(), Series.hist(), core.groupby.DataFrameGroupBy.hist(), and core.groupby.SeriesGroupBy.hist() have gained the legend argument. Set to True to show a legend in the histogram. (GH6279)

• concat() and append() now preserve extension dtypes, for example combining a nullable integer column with a numpy integer column will no longer result in object dtype but preserve the integer dtype (GH33607, GH34339, GH34095).

• read_gbq() now allows to disable progress bar (GH33360).

• read_gbq() now supports the max_results kwarg from pandas-gbq (GH34639).

• DataFrame.cov() and Series.cov() now support a new parameter ddf to support delta degrees of freedom as in the corresponding numpy methods (GH34611).

• DataFrame.to_html() and DataFrame.to_string()'s col_space parameter now accepts a list or dict to change only some specific columns' width (GH28917).

• DataFrame.to_excel() can now also write OpenOffice spreadsheet (.ods) files (GH27222)

• explode() now accepts ignore_index to reset the index, similar to pd.concat() or DataFrame.sort_values() (GH34932).

• DataFrame.to_markdown() and Series.to_markdown() now accept index argument as an alias for tabulate's showindex (GH32667)

• read_csv() now accepts string values like “0”, “0.0”, “1”, “1.0” as convertible to the nullable Boolean dtype (GH34859)

• pandas.core.window.ExponentialMovingWindow now supports a times argument that allows mean to be calculated with observations spaced by the timestamps in times (GH34839)

• DataFrame.agg() and Series.agg() now accept named aggregation for renaming the output columns/indexes. (GH26513)

• compute.use_numba now exists as a configuration option that utilizes the numba engine when available (GH33966, GH35374)

• Series.plot() now supports asymmetric error bars. Previously, if Series.plot() received a “2xN” array with error values for yerr and/or xerr, the left/lower values (first row) were mirrored, while the right/upper
values (second row) were ignored. Now, the first row represents the left/lower error values and the second row the right/upper error values. (GH9536)

**Notable bug fixes**

These are bug fixes that might have notable behavior changes.

**`MultiIndex.get_indexer` interprets `method` argument correctly**

This restores the behavior of `MultiIndex.get_indexer()` with `method='backfill'` or `method='pad'` to the behavior before pandas 0.23.0. In particular, MultiIndexes are treated as a list of tuples and padding or backfilling is done with respect to the ordering of these lists of tuples (GH29896).

As an example of this, given:

```python
In [48]: df = pd.DataFrame({
            ....:     'a': [0, 0, 0, 0],
            ....:     'b': [0, 2, 3, 4],
            ....:     'c': ['A', 'B', 'C', 'D'],
            ....:     }).set_index(['a', 'b'])

In [49]: mi_2 = pd.MultiIndex.from_product([[0], [-1, 0, 1, 3, 4, 5]])
```

The differences in reindexing `df` with `mi_2` and using `method='backfill'` can be seen here:

`pandas >= 0.23, < 1.1.0`

```python
In [1]: df.reindex(mi_2, method='backfill')
```

```
Out[1]
```

```
c
0  -1  A
   0  A
   1  D
   3  A
   4  A
   5  C
```

`pandas <0.23, >= 1.1.0`

```python
In [50]: df.reindex(mi_2, method='backfill')
```

```
c
0  -1  A
   0  A
   1  B
   3  C
   4  D
   5  NaN
```

And the differences in reindexing `df` with `mi_2` and using `method='pad'` can be seen here:

`pandas >= 0.23, < 1.1.0`

```python
In [48]: df.reindex(mi_2, method='pad')
```
In [1]: df.reindex(mi_2, method='pad')
Out[1]:
   c
0 -1 NaN
0  NaN
1  D
3  NaN
4  A
5  C

pandas < 0.23, >= 1.1.0

In [51]: df.reindex(mi_2, method='pad')
Out[51]:
   c
0 -1 NaN
0   A
1   A
3   C
4   D
5   D

Failed label-based lookups always raise KeyError

Label lookups series[key], series.loc[key] and frame.loc[key] used to raise either KeyError or TypeError depending on the type of key and type of Index. These now consistently raise KeyError (GH31867)

In [52]: ser1 = pd.Series(range(3), index=[0, 1, 2])
In [53]: ser2 = pd.Series(range(3), index=pd.date_range("2020-02-01", periods=3))

Previous behavior:

In [3]: ser1[1.5]
...  TypeError: cannot do label indexing on Int64Index with these indexers [1.5] of type float

In [4]: ser1["foo"]
...  KeyError: 'foo'

In [5]: ser1.loc[1.5]
...  TypeError: cannot do label indexing on Int64Index with these indexers [1.5] of type float

In [6]: ser1.loc["foo"]
...  KeyError: 'foo'

In [7]: ser2.loc[1]
...  TypeError: cannot do label indexing on DatetimeIndex with these indexers [1] of type int
In 

In [8]: ser2.loc[pd.Timestamp(0)]
...  
KeyError: Timestamp('1970-01-01 00:00:00')

New behavior:

In [3]: ser1[1.5]
...  
KeyError: 1.5

In [4] ser1['foo']
...  
KeyError: 'foo'

In [5]: ser1.loc[1.5]
...  
KeyError: 1.5

In [6]: ser1.loc['foo']
...  
KeyError: 'foo'

In [7]: ser2.loc[1]
...  
KeyError: 1

In [8]: ser2.loc[pd.Timestamp(0)]
...  
KeyError: Timestamp('1970-01-01 00:00:00')

Similarly, DataFrame.at() and Series.at() will raise a TypeError instead of a ValueError if an incompatible key is passed, and KeyError if a missing key is passed, matching the behavior of .loc[] (GH31722)

Failed Integer Lookups on MultiIndex Raise KeyError

Indexing with integers with a MultiIndex that has an integer-dtype first level incorrectly failed to raise KeyError when one or more of those integer keys is not present in the first level of the index (GH33539)

In [54]: idx = pd.Index(range(4))
In [55]: dti = pd.date_range("2000-01-03", periods=3)
In [56]: mi = pd.MultiIndex.from_product([idx, dti])
In [57]: ser = pd.Series(range(len(mi)), index=mi)

Previous behavior:

In [5]: ser[[5]]
Out[5]: Series([], dtype: int64)
New behavior:

```python
In [5]: ser[[5]]
...: KeyError: '5 not in index'
```

**DataFrame.merge() preserves right frame's row order**

`DataFrame.merge()` now preserves the right frame's row order when executing a right merge (GH27453)

```python
In [58]: left_df = pd.DataFrame({'animal': ['dog', 'pig'],
                        'max_speed': [40, 11]})
In [59]: right_df = pd.DataFrame({'animal': ['quetzal', 'pig'],
                        'max_speed': [80, 11]})
In [60]: left_df
Out[60]:
   animal  max_speed
0      dog       40
1      pig        11
In [61]: right_df
Out[61]:
   animal  max_speed
0  quetzal        80
1      pig        11
```

Previous behavior:

```bash
>>> left_df.merge(right_df, on=['animal', 'max_speed'], how="right")
   animal  max_speed
0      pig        11
1  quetzal        80
```

New behavior:

```python
In [62]: left_df.merge(right_df, on=['animal', 'max_speed'], how="right")
Out[62]:
   animal  max_speed
0  quetzal        80
1      pig        11
```
Assignment to multiple columns of a DataFrame when some columns do not exist

Assignment to multiple columns of a DataFrame when some of the columns do not exist would previously assign the values to the last column. Now, new columns will be constructed with the right values. (GH13658)

```
In [63]: df = pd.DataFrame({"a": [0, 1, 2], "b": [3, 4, 5]})

In [64]: df
Out[64]:
   a  b
0  0  3
1  1  4
2  2  5
```

**Previous behavior:**

```
In [3]: df["a", "c"] = 1
In [4]: df
Out[4]:
   a  b
0  1  1
1  1  1
2  1  1
```

**New behavior:**

```
In [65]: df["a", "c"] = 1

In [66]: df
Out[66]:
   a  b  c
0  1  3  1
1  1  4  1
2  1  5  1
```

Consistency across groupby reductions

Using DataFrame.groupby() with as_index=True and the aggregation nunique would include the grouping column(s) in the columns of the result. Now the grouping column(s) only appear in the index, consistent with other reductions. (GH32579)

```
In [67]: df = pd.DataFrame({"a": ["x", "x", "y", "y"], "b": [1, 1, 2, 3]})

In [68]: df
Out[68]:
   a  b
0  x  1
1  x  1
2  y  2
3  y  3
```

**Previous behavior:**
In [3]: df.groupby("a", as_index=True).nunique()
Out[4]:
   a  b
0 x 1 1
1 y 1 2

New behavior:

In [69]: df.groupby("a", as_index=True).nunique()
Out[69]:
   b
0 a
   x 1
   y 2

Using `DataFrame.groupby()` with `as_index=False` and the function `idxmax`, `idxmin`, `mad`, `nunique`, `sem`, `skew`, or `std` would modify the grouping column. Now the grouping column remains unchanged, consistent with other reductions. (GH21090, GH10355)

Previous behavior:

In [3]: df.groupby("a", as_index=False).nunique()
Out[4]:
   a  b
0 x 1 1
1 y 1 2

New behavior:

In [70]: df.groupby("a", as_index=False).nunique()
Out[70]:
   a  b
0 x 1 1
1 y 2

The method `size()` would previously ignore `as_index=False`. Now the grouping columns are returned as columns, making the result a `DataFrame` instead of a `Series`. (GH32599)

Previous behavior:

In [3]: df.groupby("a", as_index=False).size()
Out[4]:
   a
0 x 2
1 y 2
   dtype: int64

New behavior:

In [71]: df.groupby("a", as_index=False).size()
Out[71]:
   a  size
0 x 2
1 y 2
agg() lost results with as_index=False when relabeling columns

Previously agg() lost the result columns, when the as_index option was set to False and the result columns were relabeled. In this case the result values were replaced with the previous index (GH32240).

```
In [72]: df = pd.DataFrame(
  "key": ["x", "y", "z", "x", "y", "z"],
  ....:  "val": [1.0, 0.8, 2.0, 3.0, 3.6, 0.75])
   ....:
In [73]: df
Out[73]:
   key  val
0   x   1.00
1   y   0.80
2   z   2.00
3   x   3.00
4   y   3.60
5   z   0.75

Previous behavior:

In [2]: grouped = df.groupby("key", as_index=False)
In [3]: result = grouped.agg(min_val=pd.NamedAgg(column="val", aggfunc="min"))
In [4]: result
Out[4]:
   min_val
0   x
1   y
2   z

New behavior:

In [74]: grouped = df.groupby("key", as_index=False)
In [75]: result = grouped.agg(min_val=pd.NamedAgg(column="val", aggfunc="min"))
In [76]: result
Out[76]:
    key  min_val
0   x   1.00
1   y   0.80
2   z   0.75

apply and applymap on DataFrame evaluates first row/column only once

In [77]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 6]})
In [78]: def func(row):
  ....:     print(row)
  ....:     return row
  ....:
In [79]:

Previous behavior:
In [4]: df.apply(func, axis=1)
a  1
b  3
Name: 0, dtype: int64
a  1
b  3
Name: 0, dtype: int64
a  2
b  6
Name: 1, dtype: int64
Out[4]:
   a  b
0  1  3
1  2  6

New behavior:

In [79]: df.apply(func, axis=1)
a  1
b  3
Name: 0, dtype: int64
a  2
b  6
Name: 1, dtype: int64
Out[79]:
   a  b
0  1  3
1  2  6

Backwards incompatible API changes

Added check_freq argument to testing.assert_frame_equal and testing.assert_series_equal

The check_freq argument was added to testing.assert_frame_equal() and testing.assert_series_equal() in pandas 1.1.0 and defaults to True. testing.assert_frame_equal() and testing.assert_series_equal() now raise AssertionError if the indexes do not have the same frequency. Before pandas 1.1.0, the index frequency was not checked.

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH33718, GH29766, GH29723, pytables >= 3.4.3). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.15.4</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pytz</td>
<td>2015.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.7.3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.6.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>4.0.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>beautifulsoup4</td>
<td>4.6.0</td>
<td></td>
</tr>
<tr>
<td>fastparquet</td>
<td>0.3.2</td>
<td></td>
</tr>
<tr>
<td>fsspec</td>
<td>0.7.4</td>
<td></td>
</tr>
<tr>
<td>gcsfs</td>
<td>0.6.0</td>
<td>X</td>
</tr>
<tr>
<td>lxml</td>
<td>3.8.0</td>
<td></td>
</tr>
<tr>
<td>matplotlib</td>
<td>2.2.2</td>
<td></td>
</tr>
<tr>
<td>numba</td>
<td>0.46.0</td>
<td></td>
</tr>
<tr>
<td>openpyxl</td>
<td>2.5.7</td>
<td></td>
</tr>
<tr>
<td>pymysql</td>
<td>0.13.0</td>
<td></td>
</tr>
<tr>
<td>pyarrow</td>
<td>0.7.1</td>
<td></td>
</tr>
<tr>
<td>pytables</td>
<td>3.4.3</td>
<td>X</td>
</tr>
<tr>
<td>scipy</td>
<td>1.2.0</td>
<td>X</td>
</tr>
<tr>
<td>sqlalchemy</td>
<td>1.1.4</td>
<td></td>
</tr>
<tr>
<td>xarray</td>
<td>0.8.2</td>
<td></td>
</tr>
<tr>
<td>xlrd</td>
<td>1.1.0</td>
<td></td>
</tr>
<tr>
<td>xlsxwriter</td>
<td>0.9.8</td>
<td></td>
</tr>
<tr>
<td>xlwt</td>
<td>1.2.0</td>
<td></td>
</tr>
<tr>
<td>pandas-gbq</td>
<td>1.2.0</td>
<td>X</td>
</tr>
</tbody>
</table>

See Dependencies and Optional dependencies for more.

**Development changes**

- The minimum version of Cython is now the most recent bug-fix version (0.29.16) (GH33334).

**Deprecations**

- Lookups on a Series with a single-item list containing a slice (e.g. ser[[slice(0, 4)]])) are deprecated and will raise in a future version. Either convert the list to a tuple, or pass the slice directly instead (GH31333)

- DataFrame.mean() and DataFrame.median() with numeric_only=None will include datetime64 and datetime64tz columns in a future version (GH29941)

- Setting values with .loc using a positional slice is deprecated and will raise in a future version. Use .loc with labels or .iloc with positions instead (GH31840)

- DataFrame.to_dict() has deprecated accepting short names for orient and will raise in a future version (GH32515)

- Categorical.to_dense() is deprecated and will be removed in a future version, use np.asarray(cat) instead (GH32639)

- The fastpath keyword in the SingleBlockManager constructor is deprecated and will be removed in a future version (GH33092)

- Providing suffixes as a set in pandas.merge() is deprecated. Provide a tuple instead (GH33740, GH34741).

- Indexing a Series with a multi-dimensional indexer like [:, None] to return an ndarray now raises a FutureWarning. Convert to a NumPy array before indexing instead (GH27837)
**Index.is_mixed()** is deprecated and will be removed in a future version, check **Index.inferred_type** directly instead (GH32922)

Passing any arguments but the first one to **read_html()** as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573).

Passing any arguments but path_or_buf (the first one) to **read_json()** as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573).

Passing any arguments but the first two to **read_excel()** as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573).

**pandas.api.types.is_categorical()** is deprecated and will be removed in a future version; use **pandas.api.types.is_categorical_dtype()** instead (GH33385)

**Index.get_value()** is deprecated and will be removed in a future version (GH19728)

**Series.dt.week()** and **Series.dt.weekofyear()** are deprecated and will be removed in a future version, use **Series.dt.isocalendar().week()** instead (GH33595)

**DatetimeIndex.week()** and **DatetimeIndex.weekofyear()** are deprecated and will be removed in a future version, use **DatetimeIndex.isocalendar().week()** instead (GH33595)

**DatetimeArray.week()** and **DatetimeArray.weekofyear()** are deprecated and will be removed in a future version, use **DatetimeArray.isocalendar().week()** instead (GH33595)

**Series.tshift()** and **Series.tshift()** are deprecated and will be removed in a future version, use **DataFrame.shift()** and **Series.shift()** instead (GH11631)

Indexing an **Index** object with a float key is deprecated, and will raise an **IndexError** in the future. You can manually convert to an integer key instead (GH34191).

The **squeeze** keyword in **groupby()** is deprecated and will be removed in a future version (GH32380)

The **tz** keyword in **Period.to_timestamp()** is deprecated and will be removed in a future version; use **per.to_timestamp(...).tz_localize(tz)** instead (GH34522)

**DatetimeIndex.to_perioddelta()** is deprecated and will be removed in a future version. Use **index - index.to_period(freq).to_timestamp()** instead (GH34853)

**DataFrame.melt()** accepting a **value_name** that already exists is deprecated, and will be removed in a future version (GH34731)

The **center** keyword in the **DataFrame.expanding()** function is deprecated and will be removed in a future version (GH20647)
Performance improvements

- Performance improvement in `Timedelta` constructor (GH30543)
- Performance improvement in `Timestamp` constructor (GH30543)
- Performance improvement in flex arithmetic ops between `DataFrame` and `Series` with `axis=0` (GH31296)
- Performance improvement in arithmetic ops between `DataFrame` and `Series` with `axis=1` (GH33600)
- The internal index method `_shallow_copy()` now copies cached attributes over to the new index, avoiding creating these again on the new index. This can speed up many operations that depend on creating copies of existing indexes (GH28584, GH32640, GH32669)
- Significant performance improvement when creating a `DataFrame` with sparse values from scipy.sparse matrices using the `DataFrame.sparse.from_spmatrix()` constructor (GH32821, GH32825, GH32826, GH32856, GH32858).
- Performance improvement for groupby methods `first()` and `last()` (GH34178)
- Performance improvement in `factorize()` for nullable (integer and Boolean) dtypes (GH33064).
- Performance improvement when constructing `Categorical` objects (GH33921)
- Fixed performance regression in `pandas.qcut()` and `pandas.cut()` (GH33921)
- Performance improvement in reductions (sum, prod, min, max) for nullable (integer and Boolean) dtypes (GH30982, GH33261, GH33442).
- Performance improvement in arithmetic operations between two `DataFrame` objects (GH32779)
- Performance improvement in `pandas.core.groupby.RollingGroupby` (GH34052)
- Performance improvement in arithmetic operations (sub, add, mul, div) for `MultiIndex` (GH34297)
- Performance improvement in `DataFrame[bool_indexer]` when `bool_indexer` is a list (GH33924)
- Significant performance improvement of `io.formats.style.Styler.render()` with styles added with various ways such as `io.formats.style.Styler.apply()`, `io.formats.style.Styler.applymap()` or `io.formats.style.Styler.bar()` (GH19917)

Bug fixes

Categorical

- Passing an invalid `fill_value` to `Categorical.take()` raises a `ValueError` instead of `TypeError` (GH33660)
- Combining a `Categorical` with integer categories and which contains missing values with a float dtype column in operations such as `concat()` or `append()` will now result in a float column instead of an object dtype column (GH33607)
- Bug where `merge()` was unable to join on non-unique categorical indices (GH28189)
- Bug when passing categorical data to `Index` constructor along with `dtype=object` incorrectly returning a `CategoricalIndex` instead of object-dtype `Index` (GH32167)
- Bug where `Categorical` comparison operator `__ne__` would incorrectly evaluate to `False` when either element was missing (GH32276)
- `Categorical.fillna()` now accepts `Categorical` other argument (GH32420)
- Repr of `Categorical` was not distinguishing between `int` and `str` (GH33676)
Datetimelike

- Passing an integer dtype other than int64 to np.array(period_index, dtype=...) will now raise TypeError instead of incorrectly using int64 (GH32255)

- Series.to_timestamp() now raises a TypeError if the axis is not a PeriodIndex. Previously an AttributeError was raised (GH33327)

- Series.to_period() now raises a TypeError if the axis is not a DatetimeIndex. Previously an AttributeError was raised (GH33327)

- Period no longer accepts tuples for the freq argument (GH34658)

- Bug in Timestamp where constructing a Timestamp from ambiguous epoch time and calling constructor again changed the Timestamp.value() property (GH24329)

- DatetimeArray.searchsorted(), TimedeltaArray.searchsorted(), PeriodArray.searchsorted() not recognizing non-pandas scalars and incorrectly raising ValueError instead of TypeError (GH30950)

- Bug in Timestamp where constructing Timestamp with dateutil timezone less than 128 nanoseconds before daylight saving time switch from winter to summer would result in nonexistent time (GH31043)

- Bug in Period.to_timestamp(), Period.start_time() with microsecond frequency returning a timestamp one nanosecond earlier than the correct time (GH31475)

- Timestamp raised a confusing error message when year, month or day is missing (GH31200)

- Bug in DatetimeIndex constructor incorrectly accepting bool-dtype inputs (GH32668)

- Bug in DatetimeIndex.searchsorted() not accepting a list or Series as its argument (GH32762)

- Bug where PeriodIndex() raised when passed a Series of strings (GH26109)

- Bug in Timestamp arithmetic when adding or subtracting an np.ndarray with timedelta64 dtype (GH33296)

- Bug in DatetimeIndex.to_period() not inferring the frequency when called with no arguments (GH33358)

- Bug in DatetimeIndex.tz_localize() incorrectly retaining freq in some cases where the original freq is no longer valid (GH30511)

- Bug in DatetimeIndex.intersection() losing freq and timezone in some cases (GH33604)

- Bug in DatetimeIndex.get_indexer() where incorrect output would be returned for mixed datetime-like targets (GH33741)

- Bug in DatetimeIndex.addition and subtraction with some types of DateOffset objects incorrectly retaining an invalid freq attribute (GH33779)

- Bug in DatetimeIndex where setting the freq attribute on an index could silently change the freq attribute on another index viewing the same data (GH33552)

- DataFrame.min() and DataFrame.max() were not returning consistent results with Series.min() and Series.max() when called on objects initialized with empty pd.to_datetime() (GH33904)

- Bug in DatetimeIndex.intersection() and TimedeltaIndex.intersection() with results not having the correct name attribute (GH33904)

- Bug in DatetimeArray.__setitem__(), TimedeltaArray.__setitem__(), PeriodArray.__setitem__() incorrectly allowing values with int64 dtype to be silently cast (GH33717)

- Bug in subtracting TimedeltaIndex from Period incorrectly raising TypeError in some cases where it should succeed and IncompatibleFrequency in some cases where it should raise TypeError (GH33883)

- Bug in constructing a Series or Index from a read-only NumPy array with non-ns resolution which converted to object dtype instead of coercing to datetime64[ns] dtype when within the timestamp bounds (GH34843).
• The freq keyword in Period, date_range(), period_range(), pd.tseries.frequencies.to_offset() no longer allows tuples, pass as string instead (GH34703)

• Bug in DataFrame.append() when appending a Series containing a scalar tz-aware Timestamp to an empty DataFrame resulted in an object column instead of datetime64[ns, tz] dtype (GH35038)

• OutOfBoundsDatetime issues an improved error message when timestamp is out of implementation bounds. (GH32967)

• Bug in AbstractHolidayCalendar.holidays() when no rules were defined (GH31415)

• Bug in Tick comparisons raising TypeError when comparing against timedelta-like objects (GH34088)

• Bug in Tick multiplication raising TypeError when multiplying by a float (GH34486)

Timedelta

• Bug in constructing a Timedelta with a high precision integer that would round the Timedelta components (GH31354)

• Bug in dividing np.nan or None by Timedelta incorrectly returning NaT (GH31869)

• Timedelta now understands µs as an identifier for microsecond (GH32899)

• Timedelta string representation now includes nanoseconds, when nanoseconds are non-zero (GH9309)

• Bug in comparing a Timedelta object against an np.ndarray with timedelta64 dtype incorrectly viewing all entries as unequal (GH33441)

• Bug in timedelta_range() that produced an extra point on a edge case (GH30353, GH33498)

• Bug in DataFrame.resample() that produced an extra point on a edge case (GH30353, GH13022, GH33498)

• Bug in DataFrame.resample() that ignored the loffset argument when dealing with timedelta (GH7687, GH33498)

• Bug in Timedelta and pandas.to_timedelta() that ignored the unit argument for string input (GH12136)

Timezones

• Bug in to_datetime() with infer_datetime_format=True where timezone names (e.g. UTC) would not be parsed correctly (GH33133)

Numeric

• Bug in DataFrame.floordiv() with axis=0 not treating division-by-zero like Series.floordiv() (GH31271)

• Bug in to_numeric() with string argument "uint64" and errors="coerce" silently fails (GH32394)

• Bug in to_numeric() with downcast="unsigned" fails for empty data (GH32493)

• Bug in DataFrame.mean() with numeric_only=False and either datetime64 dtype or PeriodDtype column incorrectly raising TypeError (GH32426)

• Bug in DataFrame.count() with level="foo" and index level "foo" containing NaNs causes segmentation fault (GH21824)

• Bug in DataFrame.diff() with axis=1 returning incorrect results with mixed dtypes (GH32995)
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

- Bug in `DataFrame.corr()` and `DataFrame.cov()` raising when handling nullable integer columns with `pandas.NA` (GH33803)
- Bug in arithmetic operations between `DataFrame` objects with non-overlapping columns with duplicate labels causing an infinite loop (GH35194)
- Bug in `DataFrame` and `Series` addition and subtraction between object-dtype objects and `datetime64` dtype objects (GH33824)
- Bug in `Index.difference()` giving incorrect results when comparing a `Float64Index` and object `Index` (GH35217)
- Bug in `DataFrame` reductions (e.g. `df.min()`, `df.max()`) with `ExtensionArray` dtypes (GH34520, GH32651)
- `Series.interpolate()` and `DataFrame.interpolate()` now raise a `ValueError` if `limit_direction` is 'forward' or 'both' and method is 'backfill' or 'bfill' or `limit_direction` is 'backward' or 'both' and method is 'pad' or 'ffill' (GH34746)

**Conversion**

- Bug in `Series` construction from NumPy array with big-endian `datetime64` dtype (GH29684)
- Bug in `Timedelta` construction with large nanoseconds keyword value (GH32402)
- Bug in `DataFrame` construction where sets would be duplicated rather than raising (GH32582)
- The `DataFrame` constructor no longer accepts a list of `DataFrame` objects. Because of changes to NumPy, `DataFrame` objects are now consistently treated as 2D objects, so a list of `DataFrame` objects is considered 3D, and no longer acceptable for the `DataFrame` constructor (GH32289).
- Bug in `DataFrame` when initiating a frame with lists and assign columns with nested list for `MultiIndex` (GH32173)
- Improved error message for invalid construction of list when creating a new index (GH35190)

**Strings**

- Bug in the `astype()` method when converting “string” dtype data to nullable integer dtype (GH32450).
- Fixed issue where taking `min` or `max` of a `StringArray` or `Series` with `StringDtype` type would raise. (GH31746)
- Bug in `Series.str.cat()` returning NaN output when other had `Index` type (GH33425)
- `pandas.api.dtypes.is_string_dtype()` no longer incorrectly identifies categorical series as string.

**Interval**

- Bug in `IntervalArray` incorrectly allowing the underlying data to be changed when setting values (GH32782)
Indexing

- `DataFrame.xs()` now raises a `TypeError` if a `level` keyword is supplied and the axis is not a `MultiIndex`. Previously an `AttributeError` was raised (GH33610)
- Bug in slicing on a `DatetimeIndex` with a partial-timestamp dropping high-resolution indices near the end of a year, quarter, or month (GH31064)
- Bug in `PeriodIndex.get_loc()` treating higher-resolution strings differently from `PeriodIndex.get_value()` (GH31172)
- Bug in `Series.at()` and `DataFrame.at()` not matching `.loc` behavior when looking up an integer in a `Float64Index` (GH31329)
- Bug in `PeriodIndex.is_monotonic()` incorrectly returning `True` when containing leading NaT entries (GH31437)
- Bug in `DatetimeIndex.get_loc()` raising `KeyError` with converted-integer key instead of the user-passed key (GH31425)
- Bug in `Series.xs()` incorrectly returning `Timestamp` instead of `datetime64` in some object-dtype cases (GH31630)
- Bug in `DataFrame.iat()` incorrectly returning `Timestamp` instead of `datetime` in some object-dtype cases (GH32809)
- Bug in `DataFrame.at()` when either columns or index is non-unique (GH33041)
- Bug in `Series.loc()` and `DataFrame.loc()` when indexing with an integer key on a object-dtype `Index` that is not all-integers (GH31905)
- Bug in `DataFrame.iloc.__setitem__()` on a `DataFrame` with duplicate columns incorrectly setting values for all matching columns (GH15686, GH22036)
- Bug in `DataFrame.loc()` and `Series.loc()` with a `DatetimeIndex`, `TimedeltaIndex`, or `PeriodIndex` incorrectly allowing lookups of non-matching datetime-like dtypes (GH32650)
- Bug in `Series.__getitem__()` indexing with non-standard scalars, e.g. `np.dtype` (GH32684)
- Bug in `Index` constructor where an unhelpful error message was raised for NumPy scalars (GH33017)
- Bug in `DataFrame.lookup()` incorrectly raising an `AttributeError` when `frame.index` or `frame.columns` is not unique; this will now raise a `ValueError` with a helpful error message (GH33041)
- Bug in `Interval` where a `Timedelta` could not be added or subtracted from a `Timestamp` interval (GH32023)
- Bug in `DataFrame.copy()` not invalidating `_item_cache` after copy caused post-copy value updates to not be reflected (GH31784)
- Fixed regression in `DataFrame.loc()` and `Series.loc()` throwing an error when a `datetime64[ns, tz]` value is provided (GH32395)
- Bug in `Series.__getitem__()` with an integer key and a `MultiIndex` with leading integer level failing to raise `KeyError` if the key is not present in the first level (GH33355)
- Bug in `DataFrame.iloc()` when slicing a single column `DataFrame` with `ExtensionDtype` (e.g. `df.iloc[:, :1]`) returning an invalid result (GH32957)
- Bug in `DatetimeIndex.insert()` and `TimedeltaIndex.insert()` causing index `freq` to be lost when setting an element into an empty `Series` (GH33573)
- Bug in `Series.__setitem__()` with an `IntervalIndex` and a list-like key of integers (GH33473)
- Bug in `Series.__getitem__()` allowing missing labels with `np.ndarray`, `Index`, `Series` indexers but not list, these now all raise `KeyError` (GH33646)
• Bug in `DataFrame.truncate()` and `Series.truncate()` where index was assumed to be monotone increasing (GH33756)

• Indexing with a list of strings representing datetimes failed on `DatetimeIndex` or `PeriodIndex` (GH11278)

• Bug in `Series.at()` when used with a `MultiIndex` would raise an exception on valid inputs (GH26989)

• Bug in `DataFrame.loc()` with dictionary of values changes columns with dtype of int to float (GH34573)

• Bug in `Series.loc()` when used with a `MultiIndex` would raise an `IndexingError` when accessing a None value (GH34318)

• Bug in `DataFrame.reset_index()` and `Series.reset_index()` would not preserve data types on an empty `DataFrame` or `Series` with a `MultiIndex` (GH19602)

• Bug in `Series` and `DataFrame` indexing with a time key on a `DatetimeIndex` with `NaT` entries (GH35114)

### Missing

• Calling `fillna()` on an empty `Series` now correctly returns a shallow copied object. The behaviour is now consistent with `Index`, `DataFrame` and a non-empty `Series` (GH32543).

• Bug in `Series.replace()` when argument `to_replace` is of type dict/list and is used on a `Series` containing `<NA>` was raising a `TypeError`. The method now handles this by ignoring `<NA>` values when doing the comparison for the replacement (GH32621)

• Bug in `any()` and `all()` incorrectly returning `<NA>` for all False or all True values using the nullable Boolean dtype and with `skipna=False` (GH32523)

• Clarified documentation on interpolate with `method=akima`. The `der` parameter must be scalar or `None` (GH33426)

• `DataFrame.interpolate()` uses the correct axis convention now. Previously interpolating along columns lead to interpolation along indices and vice versa. Furthermore interpolating with methods `pad`, `ffill`, `bfill` and `backfill` are identical to using these methods with `DataFrame.fillna()` (GH12918, GH29146)

• Bug in `DataFrame.interpolate()` when called on a `DataFrame` with column names of string type was throwing a `ValueError`. The method is now independent of the type of the column names (GH33956)

• Passing `NA` into a format string using format specs will now work. For example `"{:.1f}".format(pd.NA) would previously raise a `ValueError`, but will now return the string "<NA>" (GH34740)

• Bug in `Series.map()` not raising on invalid `na_action` (GH32815)

### MultiIndex

• `DataFrame.swaplevels()` now raises a `TypeError` if the axis is not a `MultiIndex`. Previously an `AttributeError` was raised (GH31126)

• Bug in `DataFrame.loc()` when used with a `MultiIndex`. The returned values were not in the same order as the given inputs (GH22797)

```python
In [80]: df = pd.DataFrame(np.arange(4),
   ....:     index=[["a", "a", "b", "b"], [1, 2, 1, 2]])
   ....:

# Rows are now ordered as the requested keys
In [81]: df.loc[['b', 'a'], [2, 1]], :
```

(continues on next page)
Out[81]:
    0
   b 2 3
    1 2
   a 2 1
    1 0

• Bug in `MultiIndex.intersection()` was not guaranteed to preserve order when `sort=False`. (GH31325)
• Bug in `DataFrame.truncate()` was dropping `MultiIndex` names. (GH34564)

In [82]: left = pd.MultiIndex.from_arrays([["b", "a"], [2, 1]])
In [83]: right = pd.MultiIndex.from_arrays([["a", "b", "c"], [1, 2, 3]])
# Common elements are now guaranteed to be ordered by the left side
In [84]: left.intersection(right, sort=False)
Out[84]:
MultiIndex([('b', 2),
            ('a', 1)],
           names=['a', 'b'])

• Bug when joining two `MultiIndex` without specifying level with different columns. Return-indexers parameter was ignored. (GH34074)

IO

• Passing a set as names argument to `pandas.read_csv()`, `pandas.read_table()`, or `pandas.read_fwf()` will raise `ValueError`: Names should be an ordered collection. (GH34946)
• Bug in print-out when `display.precision` is zero. (GH30320)
• `read_csv()` will now raise a `ValueError` when the arguments `header` and `prefix` both are not `None`. (GH27394)
• Bug in `DataFrame.to_json()` was raising `NotFoundError` when `path_or_buf` was an S3 URI (GH28375)
• Bug in `DataFrame.to_parquet()` overwriting pyarrow’s default for `coerce_timestamps`; following pyarrow’s default allows writing nanosecond timestamps with version="2.0" (GH31652).
• Bug in `read_csv()` was raising `TypeError` when `sep=None` was used in combination with `comment` keyword (GH31396)
• Bug in HDFStore that caused it to set to int64 the dtype of a datetime64 column when reading a `DataFrame` in Python 3 from fixed format written in Python 2 (GH31750)
• `read_sas()` now handles dates and datetimes larger than `Timestamp.max` returning them as `datetime` objects (GH20927)
• Bug in `DataFrame.to_json()` where Timedelta objects would not be serialized correctly with `date_format="iso"` (GH28256)
• `read_csv()` will raise a `ValueError` when the column names passed in `parse_dates` are missing in the `Dataframe` (GH31251)
• Bug in `read_excel()` where a UTF-8 string with a high surrogate would cause a segmentation violation (GH23809)
• Bug in `read_csv()` was causing a file descriptor leak on an empty file (GH31488)
• Bug in `read_csv()` was causing a segfault when there were blank lines between the header and data rows (GH28071)
• Bug in `read_csv()` was raising a misleading exception on a permissions issue (GH23784)
• Bug in `read_csv()` was raising an IndexError when header=None and two extra data columns
• Bug in `read_sas()` was raising an AttributeError when reading files from Google Cloud Storage (GH33069)
• Bug in `DataFrame.to_sql()` where an AttributeError was raised when saving an out of bounds date (GH26761)
• Bug in `read_excel()` did not correctly handle multiple embedded spaces in OpenDocument text cells. (GH32207)
• Bug in `read_json()` was raising TypeError when reading a list of Booleans into a Series. (GH31464)
• Bug in pandas.io.json.json_normalize() where location specified by record_path doesn’t point to an array. (GH26284)
• `pandas.read_hdf()` has a more explicit error message when loading an unsupported HDF file (GH9539)
• Bug in `read_feather()` was raising an ArrowIOError when reading an s3 or http file path (GH29055)
• Bug in `to_excel()` could not handle the column name `render` and was raising an KeyError (GH34331)
• Bug in `execute()` was raising a ProgrammingError for some DB-API drivers when the SQL statement contained the % character and no parameters were present (GH34211)
• Bug in StataReader() which resulted in categorical variables with different dtypes when reading data using an iterator. (GH31544)
• `HDFStore.keys()` has now an optional include parameter that allows the retrieval of all native HDF5 table names (GH29916)
• TypeError exceptions raised by `read_csv()` and `read_table()` were showing as parser_f when an unexpected keyword argument was passed (GH25648)
• Bug in `read_excel()` for ODS files removes 0.0 values (GH27222)
• Bug in `ujson.encode()` was raising an OverflowError with numbers larger than sys.maxsize (GH34395)
• Bug in `HDFStore.append_to_multiple()` was raising a ValueError when the min_itemsize parameter is set (GH11238)
• Bug in `create_table()` now raises an error when `column` argument was not specified in `data_columns` on input (GH28156)
• `read_json()` now could read line-delimited json file from a file url while lines and chunksize are set.
• Bug in `DataFrame.to_sql()` when reading DataFrames with `-np.inf` entries with MySQL now has a more explicit Value Error (GH34431)
• Bug where capitalised files extensions were not decompressed by read_* functions (GH35164)
• Bug in `read_excel()` that was raising a TypeError when header=None and index_col is given as a list (GH31783)
• Bug in `read_excel()` where datetime values are used in the header in a MultiIndex (GH34748)
• `read_excel()` no longer takes **kwds arguments. This means that passing in the keyword argument chunksize now raises a TypeError (previously raised a `NotImplementedError`), while passing in the keyword argument encoding now raises a TypeError (GH34464)

• Bug in `DataFrame.to_records()` was incorrectly losing timezone information in timezone-aware `datetime64` columns (GH32535)

**Plotting**

• `DataFrame.plot()` for line/bar now accepts color by dictionary (GH8193).

• Bug in `DataFrame.plot.hist()` where weights are not working for multiple columns (GH33173)

• Bug in `DataFrame.boxplot()` and `DataFrame.plot.boxplot()` lost color attributes of `medianprops`, `whiskerprops`, `capprops` and `boxprops` (GH30346)

• Bug in `DataFrame.hist()` where the order of column argument was ignored (GH29235)

• Bug in `DataFrame.plot.scatter()` that when adding multiple plots with different `cmap`, colorbars always use the first `cmap` (GH33389)

• Bug in `DataFrame.plot.scatter()` was adding a colorbar to the plot even if the argument `c` was assigned to a column containing color names (GH34316)

• Bug in `pandas.plotting.bootstrap_plot()` was causing cluttered axes and overlapping labels (GH34905)

• Bug in `DataFrame.plot.scatter()` caused an error when plotting variable marker sizes (GH32904)

**GroupBy/resample/rolling**

• Using a `pandas.api.indexers.BaseIndexer` with `count`, `min`, `max`, `median`, `skew`, `cov`, `corr` will now return correct results for any monotonic `pandas.api.indexers.BaseIndexer` descendant (GH32865)

• `DataFrameGroupBy.mean()` and `SeriesGroupby.mean()` (and similarly for `median()`, `std()` and `var()`) now raise a TypeError if a non-accepted keyword argument is passed into it. Previously an UnsupportedFunctionCall was raised (AssertionError if `min_count` passed into `median()`) (GH31485)

• Bug in `GroupBy.apply()` raises ValueError when the `by` axis is not sorted, has duplicates, and the applied func does not mutate passed in objects (GH30667)

• Bug in `DataFrameGroupBy.transform()` produces an incorrect result with transformation functions (GH30918)

• Bug in `GroupBy.transform()` was returning the wrong result when grouping by multiple keys of which some were categorical and others not (GH32494)

• Bug in `GroupBy.count()` causes segmentation fault when grouped-by columns contain NaNs (GH32841)

• Bug in `DataFrame.groupby()` and `Series.groupby()` produces inconsistent type when aggregating Boolean `Series` (GH32894)

• Bug in `DataFrameGroupBy.sum()` and `SeriesGroupBy.sum()` where a large negative number would be returned when the number of non-null values was below `min_count` for nullable integer dtypes (GH32861)

• Bug in `SeriesGroupBy.quantile()` was raising on nullable integers (GH33136)

• Bug in `DataFrame.resample()` where an AmbiguousTimeError would be raised when the resulting timezone aware `DatetimeIndex` had a DST transition at midnight (GH25758)

• Bug in `DataFrame.groupby()` where a ValueError would be raised when grouping by a categorical column with read-only categories and `sort=False` (GH33410)
• Bug in GroupBy.agg(), GroupBy.transform(), and GroupBy.resample() where subclasses are not preserved (GH28330)

• Bug in SeriesGroupBy.agg() where any column name was accepted in the named aggregation of SeriesGroupBy previously. The behaviour now allows only str and callables else would raise TypeError. (GH34422)

• Bug in DataFrame.groupby() lost the name of the Index when one of the agg keys referenced an empty list (GH32580)

• Bug in Rolling.apply() where center=True was ignored when engine='numba' was specified (GH34784)

• Bug in DataFrame.ewm.cov() was throwing AssertionError for MultiIndex inputs (GH34440)

• Bug in core.groupby.DataFrameGroupBy.quantile() raised TypeError for non-numeric types rather than dropping the columns (GH27892)

• Bug in core.groupby.DataFrameGroupBy.transform() when func='nunique' and columns are of type datetime64, the result would also be of type datetime64 instead of int64 (GH35109)

• Bug in DataFrame.groupby() raising an AttributeError when selecting a column and aggregating with as_index=False (GH35246)

• Bug in DataFrameGroupBy.first() and DataFrameGroupBy.last() that would raise an unnecessary ValueError when grouping on multiple Categoricals (GH34951)

**Reshaping**

• Bug effecting all numeric and Boolean reduction methods not returning subclassed data type. (GH25596)

• Bug in DataFrame.pivot_table() when only MultiIndexed columns is set (GH17038)

• Bug in DataFrame.unstack() and Series.unstack() can take tuple names in MultiIndexed data (GH19966)

• Bug in DataFrame.pivot_table() when margin is True and only column is defined (GH31016)

• Fixed incorrect error message in DataFrame.pivot() when columns is set to None. (GH30924)

• Bug in crosstab() when inputs are two Series and have tuple names, the output will keep a dummy MultiIndex as columns. (GH18321)

• DataFrame.pivot() can now take lists for index and columns arguments (GH21425)

• Bug in concat() where the resulting indices are not copied when copy=True (GH29879)

• Bug in SeriesGroupBy.aggregate() was resulting in aggregations being overwritten when they shared the same name (GH30880)

• Bug where Index.astype() would lose the name attribute when converting from Float64Index to Int64Index, or when casting to an ExtensionArray dtype (GH32013)

• Series.append() will now raise a TypeError when passed a DataFrame or a sequence containing DataFrame (GH31413)

• DataFrame.replace() and Series.replace() will raise a TypeError if to_replace is not an expected type. Previously the replace would fail silently (GH31413)

• Bug on inplace operation of a Series that was adding a column to the DataFrame from where it was originally dropped from (using inplace=True) (GH30484)

• Bug in DataFrame.apply() where callback was called with Series parameter even though raw=True requested. (GH32423)
• Bug in `DataFrame.pivot_table()` losing timezone information when creating a `MultiIndex` level from a column with timezone-aware dtype (GH32558)

• Bug in `concat()` where when passing a non-dict mapping as `objs` would raise a `TypeError` (GH32863)

• `DataFrame.agg()` now provides more descriptive `SpecificationError` message when attempting to aggregate a non-existent column (GH32755)

• Bug in `DataFrame.unstack()` when `MultiIndex` columns and `MultiIndex` rows were used (GH32624, GH24729 and GH28306)

• Appending a dictionary to a `DataFrame` without passing `ignore_index=True` will raise `TypeError`: Can only append a dict if `ignore_index=True` or if the :class:`Series` has a name (GH30871)

• Bug in `DataFrame.corrwith()`, `DataFrame.memory_usage()`, `DataFrame.dot()`, `DataFrame.idxmin()`, `DataFrame.idxmax()`, `DataFrame.duplicated()`, `DataFrame.isin()`, `DataFrame.count()`, `Series.explode()`, `Series.asof()` and `DataFrame.asof()` not returning subclassed types. (GH31331)

• Bug in `concat()` was not allowing for concatenation of `DataFrame` and `Series` with duplicate keys (GH33654)

• Bug in `cut()` raised an error when the argument `labels` contains duplicates (GH33141)

• Ensure only named functions can be used in `eval()` (GH32460)

• Bug in `DataFrame.aggregate()` and `Series.aggregate()` was causing a recursive loop in some cases (GH34224)

• Fixed bug in `melt()` where melting `MultiIndex` columns with `col_level > 0` would raise a `KeyError` on `id_vars` (GH34129)

• Bug in `Series.where()` with an empty `Series` and empty `cond` having non-bool dtype (GH34592)

• Fixed regression where `DataFrame.apply()` would raise `ValueError` for elements with S dtype (GH34529)

**Sparse**

• Creating a `SparseArray` from timezone-aware dtype will issue a warning before dropping timezone information, instead of doing so silently (GH32501)

• Bug in `arrays.SparseArray.from_spmatrix()` wrongly read scipy sparse matrix (GH31991)

• Bug in `Series.sum()` with `SparseArray` raised a `TypeError` (GH25777)

• Bug where `DataFrame` containing an all-sparse `SparseArray` filled with NaN when indexed by a list-like (GH27781, GH29563)

• The repr of `SparseDtype` now includes the repr of its `fill_value` attribute. Previously it used `fill_value`'s string representation (GH34352)

• Bug where empty `DataFrame` could not be cast to `SparseDtype` (GH33113)

• Bug in `arrays.SparseArray()` was returning the incorrect type when indexing a sparse dataframe with an iterable (GH34526, GH34540)
ExtensionArray

- Fixed bug where `Series.value_counts()` would raise on empty input of Int64 dtype (GH33317)
- Fixed bug in `concat()` when concatenating DataFrame objects with non-overlapping columns resulting in object-dtype columns rather than preserving the extension dtype (GH27692, GH33027)
- Fixed bug where `StringArray.isna()` would return False for NA values when `pandas.options.mode.use_inf_as_na` was set to True (GH3655)
- Fixed bug in `Series` construction with EA dtype and index but no data or scalar data fails (GH26469)
- Fixed bug that caused `Series.__repr__()` to crash for extension types whose elements are multidimensional arrays (GH33770).
- Fixed bug where `Series.update()` would raise a `ValueError` for ExtensionArray dtypes with missing values (GH33980)
- Fixed bug where `StringArray.memory_usage()` was not implemented (GH33963)
- Fixed bug where `DataFrameGroupBy()` would ignore the `min_count` argument for aggregations on nullable Boolean dtypes (GH34051)
- Fixed bug where the constructor of DataFrame with dtype='string' would fail (GH27953, GH33623)
- Bug where DataFrame column set to scalar extension type was considered an object type rather than the extension type (GH34832)
- Fixed bug in `IntegerArray.astype()` to correctly copy the mask as well (GH34931).

Other

- Set operations on an object-dtype `Index` now always return object-dtype results (GH31401)
- Fixed `pandas.testing.assert_series_equal()` to correctly raise if the `left` argument is a different subclass with `check_series_type=True` (GH32670).
- Getting a missing attribute in a `DataFrame.query()` or `DataFrame.eval()` string raises the correct AttributeError (GH32408)
- Fixed bug in `pandas.testing.assert_series_equal()` where dtypes were checked for Interval and ExtensionArray operands when `check_dtypes` was False (GH32747)
- Bug in DataFrame.__dir__() caused a segfault when using unicode surrogates in a column name (GH25509)
- Bug in DataFrame.equals() and Series.equals() in allowing subclasses to be equal (GH34402).

Contributors

A total of 368 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- 3vts +
- A Brooks +
- Abbie Popa +
- Achmad Syarif Hidayatullah +
- Adam W Bagaskarta +
• Adrian Mastronardi +
• Aidan Montare +
• Akbar Septriyansyah +
• Akos Furton +
• Alejandro Hall +
• Alex Hall +
• Alex Itkes +
• Alex Kirko
• Ali McMaster +
• Alvaro Aleman +
• Amy Graham +
• Andrew Schonfeld +
• Andrew Shumanskiy +
• Andrew Witeseska +
• Angela Ambroz
• Anjali Singh +
• Anna Daglis
• Anthony Milbourne +
• Antony Lee +
• Ari Sosnovsky +
• Arkadeep Adhikari +
• Arunim Samudra +
• Ashkan +
• Ashwin Prakash Nalwade +
• Ashwin Srinath +
• Atsushi Nukariya +
• Ayappan +
• Ayla Khan +
• Bart +
• Bart Broere +
• Benjamin Beier Liu +
• Benjamin Fischer +
• Bharat Raghunathan
• Bradley Dice +
• Brendan Sullivan +
• Brian Strand +
• Carsten van Weelden +
• Chamoun Saoma +
• ChrisRobo +
• Christian Chwala
• Christopher Whelan
• Christos Petropoulos +
• Chuanzhu Xu
• CloseChoice +
• Clément Robert +
• CuylenE +
• DanBasson +
• Daniel Saxton
• Danilo Horta +
• DavallhamHaeruzaman +
• Dave Hirschfeld
• Dave Hughes
• David Rouquet +
• David S +
• Deepyaman Datta
• Dennis Bakhuis +
• Derek McCammond +
• Devjeet Roy +
• Diane Trout
• Dina +
• Dom +
• Drew Seibert +
• EdAbati
• Emiliano Jordan +
• Erfan Nariman +
• Eric Groszman +
• Erik Hasse +
• Erkam Uyanik +
• Evan D +
• Evan Kanter +
• Fangchen Li +
• Farhan Reynaldo +
- Farhan Reynaldo Hutabarat +
- Florian Jetter +
- Fred Reiss +
- GYHHAHA +
- Gabriel Moreira +
- Gabriel Tutui +
- Galuh Sahid
- Gaurav Chauhan +
- George Hartzell +
- Gim Seng +
- Giovanni Lanzani +
- Gordon Chen +
- Graham Wetzler +
- Guillaume Lemaitre
- Guillem Sánchez +
- HH-MWB +
- Harshvardhan Bachina
- How Si Wei
- Ian Eaves
- Iqrar Agalosi Nureyza +
- Irv Lustig
- Iva Laginja +
- JDkuba
- Jack Greisman +
- Jacob Austin +
- Jacob Deppen +
- Jacob Peacock +
- Jake Tae +
- Jake Vanderplas +
- James Cobon-Kerr
- Jan Červenka +
- Jan Škoda
- Jane Chen +
- Jean-Francois Zinque +
- Jeanderson Barros Candido +
- Jeff Reback
• Jered Dominguez-Trujillo +
• Jeremy Schendel
• Jesse Farnham
• Jiaxiang
• Jihwan Song +
• Joaquim L. Viegas +
• Joel Nothman
• John Bodley +
• John Paton +
• Jon Thielen +
• Joris Van den Bossche
• Jose Manuel Martí +
• Joseph Gulian +
• Josh Dimarsky
• Joy Bhalla +
• João Veiga +
• Julian de Ruiter +
• Justin Essert +
• Justin Zheng
• KD-dev-lab +
• Kaiqi Dong
• Karthik Mathur +
• Kaushal Rohit +
• Kee Chong Tan
• Ken Mankoff +
• Kendall Masse
• Kenny Huynh +
• Ketan +
• Kevin Anderson +
• Kevin Bowey +
• Kevin Sheppard
• Kilian Lieret +
• Koki Nishihara +
• Krishna Chivukula +
• KrishnaSai2020 +
• Lesley +
• Lewis Cowles +
• Linda Chen +
• Linxiao Wu +
• Lucca Delchiaro Costabile +
• MBrouns +
• Mabel Villalba
• Mabroor Ahmed +
• Madhuri Palanivelu +
• Mak Sze Chun
• Malcolm +
• Marc Garcia
• Marco Gorelli
• Marian Denes +
• Martin Bjeldbak Madsen +
• Martin Durant +
• Martin Fleischmann +
• Martin Jones +
• Martin Winkel
• Martina Oefelein +
• Marvzinc +
• María Marino +
• Matheus Cardoso +
• Mathis Felardos +
• Matt Roeschke
• Matteo Felici +
• Matteo Santamaria +
• Matthew Roeschke
• Matthias Bussonnier
• Max Chen
• Max Halford +
• Mayank Bisht +
• Megan Thong +
• Michael Marino +
• Miguel Marques +
• Mike Kutzma
• Mohammad Hasnain Mohsin Rajan +
pandas: powerful Python data analysis toolkit, Release 1.4.2

• Mohammad Jafar Mashhadi +
• MomIsBestFriend
• Monica +
• Natalie Jann
• Nate Armstrong +
• Nathanael +
• Nick Newman +
• Nico Schlömer +
• Niklas Weber +
• ObliviousParadigm +
• Olga Lyashevska +
• OlivierLuG +
• Pandas Development Team
• Parallels +
• Patrick +
• Patrick Cando +
• Paul Lilley +
• Paul Sanders +
• Pearcekieser +
• Pedro Larroy +
• Pedro Reys
• Peter Bull +
• Peter Steinbach +
• Phan Duc Nhat Minh +
• Phil Kirlin +
• Pterre-Yves Bourguignon +
• Piotr Kasprzyk +
• Piotr Niełacny +
• Prakhar Pandey
• Prashant Anand +
• Puneetha Pai +
• Quang Nguyen +
• Rafael Jaimes III +
• Rafif +
• RaisaDZ +
• Rakshit Naidu +
• Ram Rachum +
• Red +
• Ricardo Alanis +
• Richard Shadrach +
• Rik-de-Kort
• Robert de Vries
• Robin to Roxel +
• Roger Erens +
• Rohith295 +
• Roman Yurchak
• Ror +
• Rushabh Vasani
• Ryan
• Ryan Nazareth
• SAI SRAVAN MEDICHERLA +
• SHUBH CHATTERJEE +
• Sam Cohan
• Samira-g-js +
• Sandu Ursu +
• Sang Agung +
• SanthoshBala18 +
• Sasidhar Kasturi +
• SatheeshKumar Mohan +
• Saul Shanabrook
• Scott Gigante +
• Sebastian Berg +
• Sebastián Vanrell
• Sergei Chipiga +
• Sergey +
• ShilpaSugan +
• Simon Gibbons
• Simon Hawkins
• Simon Legner +
• Soham Tiwari +
• Song Wenhao +
• Souvik Mandal
• alm +
• andhikayusup +
• andresmcneill +
• avinashpancham +
• benabel +
• bernie gray +
• biddwan09 +
• brock +
• chris-b1
• cleconte987 +
• dan1261 +
• david-cortes +
• davidwales +
• dequadras +
• dhuettenmoser +
• dilex42 +
• elmonsomiat +
• epizzigoni +
• fjetter
• gabrielvf1 +
• gdex1 +
• gfyoung
• guru kiran +
• h-vishal
• iamshwin
• jamin-aws-ospo +
• jbrockmendel
• jfcorbett +
• jnecus +
• kernc
• kota matsuoka +
• kylekeppler +
• leandermaben +
• link2xt +
• manoj_koneni +
• marydmit +
• masterpiga +
• maxime.song +
• mglasder +
• moaraccounts +
• mproszewska
• neilkg
• nrebena
• ossdev07 +
• paihu
• pan Jacek +
• partev +
• patrick +
• pedrooa +
• pizzathief +
• proost
• pvanhauw +
• rbenes
• rebecca-palmer
• rhshadrach +
• rjfs +
• s-scherrer +
• sage +
• sagungrp +
• salem3358 +
• saloni30 +
• smartswdeveloper +
• smartvinnetou +
• themien +
• timhunderwood +
• tolhassianipar +
• tonywu1999
• tsvikas
• tv3141
• venkateshdatta1993 +
• vivikelapoutre +
• willbowditch +
5.5 Version 1.0

5.5.1 What’s new in 1.0.5 (June 17, 2020)

These are the changes in pandas 1.0.5. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fix regression in read_parquet() when reading from file-like objects (GH34467).
- Fix regression in reading from public S3 buckets (GH34626).

Note this disablesthe ability to read Parquet files from directories on S3 again (GH26388, GH34632), which was added in the 1.0.4 release, but is now targeted for pandas 1.1.0.

- Fixed regression in replace() raising an AssertionError when replacing values in an extension dtype with values of a different dtype (GH34530)

Bug fixes

- Fixed building from source with Python 3.8 fetching the wrong version of NumPy (GH34666)

Contributors

A total of 8 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Joris Van den Bossche
- MeeseeksMachine
- Natalie Jann +
- Pandas Development Team
- Simon Hawkins
- Tom Augspurger
- William Ayd
- alimcmaster1
5.5.2 What’s new in 1.0.4 (May 28, 2020)

These are the changes in pandas 1.0.4. See Release notes for a full changelog including other versions of pandas.

**Fixed regressions**

- Fix regression where `Series.isna()` and `DataFrame.isna()` would raise for categorical dtype when `pandas.options.mode.use_inf_as_na` was set to True (GH33594)
- Fix regression in `GroupBy.first()` and `GroupBy.last()` where None is not preserved in object dtype (GH32800)
- Fix regression in DataFrame reductions using `numeric_only=True` and ExtensionArrays (GH33256).
- Fix performance regression in `memory_usage(deep=True)` for object dtype (GH33012)
- Fix regression where `Categorical.replace()` would replace with NaN whenever the new value and replace-value were equal (GH33288)
- Fix regression where an ordered `Categorical` containing only NaN values would raise rather than returning NaN when taking the minimum or maximum (GH33450)
- Fix regression in `DataFrameGroupBy.agg()` with dictionary input losing ExtensionArray dtypes (GH32194)
- Fix to preserve the ability to index with the “nearest” method with xarray’s CFTimeIndex, an `Index` subclass (pydata/xarray#3751, GH32905).
- Fix regression in `DataFrame.describe()` raising `TypeError: unhashable type: 'dict'` (GH32409)
- Fix regression in `DataFrame.replace()` casts columns to object dtype if items in `to_replace` not in values (GH32988)
- Fix regression in `Series.groupby()` would raise `ValueError` when grouping by `PeriodIndex` level (GH34010)
- Fix regression in `GroupBy.rolling.apply()` ignores args and kwargs parameters (GH33433)
- Fix regression in error message with `np.min` or `np.max` on unordered `Categorical` (GH33115)
- Fix regression in `DataFrame.loc()` and `Series.loc()` throwing an error when a datetime64[ns, tz] value is provided (GH32395)

**Bug fixes**

- Bug in `SeriesGroupBy.first()`, `SeriesGroupBy.last()`, `SeriesGroupBy.min()`, and `SeriesGroupBy.max()` returning floats when applied to nullable Booleans (GH33071)
- Bug in `Rolling.min()` and `Rolling.max()`: Growing memory usage after multiple calls when using a fixed window (GH30726)
- Bug in `to_parquet()` was not raising `PermissionError` when writing to a private s3 bucket with invalid creds. (GH27679)
- Bug in `to_csv()` was silently failing when writing to an invalid s3 bucket. (GH32486)
- Bug in `read_parquet()` was raising a `FileNotFoundException` when passed an s3 directory path. (GH26388)
- Bug in `to_parquet()` was throwing an `AttributeError` when writing a partitioned parquet file to s3 (GH27596)
- Bug in `GroupBy.quantile()` causes the quantiles to be shifted when the by axis contains NaN (GH33200, GH33569)
Contributors

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Daniel Saxton
- JDKuba +
- Joris Van den Bossche
- Kaiqi Dong
- Mabel Villalba
- MeeseeksMachine
- MomIsBestFriend
- Pandas Development Team
- Simon Hawkins
- Spencer Clark +
- Tom Augspurger
- Vikas Pandey +
- alimcmaster1
- h-vishal +
- jbrockmendel
- mproszewska +
- neilk +
- rebecca-palmer +

5.5.3 What’s new in 1.0.3 (March 17, 2020)

These are the changes in pandas 1.0.3. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in `resample.agg` when the underlying data is non-writeable (GH31710)
- Fixed regression in `DataFrame` exponentiation with reindexing (GH32685)

Bug fixes

Contributors

A total of 5 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- MeeseeksMachine
- Pandas Development Team
- Tom Augspurger
5.5.4 What’s new in 1.0.2 (March 12, 2020)

These are the changes in pandas 1.0.2. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

Groupby

- Fixed regression in `groupby(..).agg()` which was failing on frames with `MultiIndex` columns and a custom function (GH31777)
- Fixed regression in `groupby(..).rolling(..).apply()` (RollingGroupby) where the `raw` parameter was ignored (GH31754)
- Fixed regression in `rolling(..).corr()` when using a time offset (GH31789)
- Fixed regression in `groupby(..).nunique()` which was modifying the original values if `NaN` values were present (GH31950)
- Fixed regression in `DataFrame.groupby` raising a `ValueError` from an internal operation (GH31802)
- Fixed regression in `groupby(..).agg()` calling a user-provided function an extra time on an empty input (GH31760)

I/O

- Fixed regression in `read_csv()` in which the `encoding` option was not recognized with certain file-like objects (GH31819)
- Fixed regression in `DataFrame.to_excel()` when the `columns` keyword argument is passed (GH31677)
- Fixed regression in `ExcelFile` where the stream passed into the function was closed by the destructor. (GH31467)
- Fixed regression where `read_pickle()` raised a `UnicodeDecodeError` when reading a py27 pickle with `MultiIndex` column (GH31988).

Reindexing/alignment

- Fixed regression in `Series.align()` when `other` is a `DataFrame` and `method` is not `None` (GH31785)
- Fixed regression in `DataFrame.reindex()` and `Series.reindex()` when reindexing with (tz-aware) index and `method=nearest` (GH26683)
- Fixed regression in `DataFrame.reindex_like()` on a `DataFrame` subclass raised an `AssertionError` (GH31925)
- Fixed regression in `DataFrame` arithmetic operations with mis-matched columns (GH31623)

Other

- Fixed regression in joining on `DatetimeIndex` or `TimedeltaIndex` to preserve `freq` in simple cases (GH32166)
- Fixed regression in `Series.shift()` with `datetime64` dtype when passing an integer `fill_value` (GH32591)
- Fixed regression in the repr of an object-dtype `Index` with bools and missing values (GH32146)
Indexing with nullable boolean arrays

Previously indexing with a nullable Boolean array containing NA would raise a ValueError, however this is now permitted with NA being treated as False. (GH31503)

```python
In [1]: s = pd.Series([1, 2, 3, 4])
In [2]: mask = pd.array([True, True, False, None], dtype="boolean")
In [3]: s
Out[3]:
0  1
1  2
2  3
3  4
dtype: int64
In [4]: mask
Out[4]:
<BooleanArray>
[True, True, False, <NA>]
Length: 4, dtype: boolean
```

```
pandas 1.0.0-1.0.1
>>> s[mask]
Traceback (most recent call last):
  ...
ValueError: cannot mask with array containing NA / NaN values
```

```
pandas 1.0.2
In [5]: s[mask]
Out[5]:
0  1
1  2
dtype: int64
```

Bug fixes

Datetimelike

- Bug in `Series.astype()` not copying for tz-naive and tz-aware datetime64 dtype (GH32490)
- Bug where `to_datetime()` would raise when passed `pd.NA` (GH32213)
- Improved error message when subtracting two `Timestamp` that result in an out-of-bounds `Timedelta` (GH31774)

Categorical

- Fixed bug where `Categorical.from_codes()` improperly raised a `ValueError` when passed nullable integer codes. (GH31779)
- Fixed bug where `Categorical()` constructor would raise a `TypeError` when given a numpy array containing `pd.NA`. (GH31927)
pandas: powerful Python data analysis toolkit, Release 1.4.2

• Bug in *Categorical* that would ignore or crash when calling *Series.replace()* with a list-like *to_replace* (GH31720)

I/O

• Using *pd.NA* with *DataFrame.to_json()* now correctly outputs a null value instead of an empty object (GH31615)
• Bug in *pandas.json_normalize()* when value in meta path is not iterable (GH31507)
• Fixed pickling of *pandas.NA*. Previously a new object was returned, which broke computations relying on *NA* being a singleton (GH31847)
• Fixed bug in parquet roundtrip with nullable unsigned integer dtypes (GH31896).

Experimental dtypes

• Fixed bug in *DataFrame.convert_dtypes()* for columns that were already using the "string" dtype (GH31731).
• Fixed bug in *DataFrame.convert_dtypes()* for series with mix of integers and strings (GH32117)
• Fixed bug in *DataFrame.convert_dtypes()* where *BooleanDtype* columns were converted to *Int64* (GH32287)
• Fixed bug in setting values using a slice indexer with string dtype (GH31772)
• Fixed bug where *pandas.core.groupby.GroupBy.first()* and *pandas.core.groupby.GroupBy.last()* would raise a *TypeError* when groups contained *pd.NA* in a column of object dtype (GH32123)
• Fixed bug where *DataFrameGroupBy.mean()* , *DataFrameGroupBy.median()* , *DataFrameGroupBy.var()* , and *DataFrameGroupBy.std()* would raise a *TypeError* on *Int64* dtype columns (GH32219)

Strings

• Using *pd.NA* with *Series.str.repeat()* now correctly outputs a null value instead of raising error for vector inputs (GH31632)

Rolling

• Fixed rolling operations with variable window (defined by time duration) on decreasing time index (GH32385).

Contributors

A total of 25 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Anna Daglis +
• Daniel Saxton
• Irv Lustig
• Jan Škoda
• Joris Van den Bossche
• Justin Zheng
• Kaiqi Dong
• Kendall Masse
• Marco Gorelli
• Matthew Roeschke
5.5.5 What’s new in 1.0.1 (February 5, 2020)

These are the changes in pandas 1.0.1. See Release notes for a full changelog including other versions of pandas.

Fixed regressions

- Fixed regression in DataFrame setting values with a slice (e.g. df[-4:] = 1) indexing by label instead of position (GH31469)
- Fixed regression when indexing a Series or DataFrame indexed by DatetimeIndex with a slice containing a datetime.date (GH31501)
- Fixed regression in DataFrame.__setitem__ raising an AttributeError with a MultiIndex and a non-monotonic indexer (GH31449)
- Fixed regression in Series multiplication when multiplying a numeric Series with >10000 elements with a timedelta-like scalar (GH31457)
- Fixed regression in .groupby().agg() raising an AssertionError for some reductions like min on object-dtype columns (GH31522)
- Fixed regression in .groupby() aggregations with categorical dtype using Cythonized reduction functions (e.g. first) (GH31450)
- Fixed regression in GroupBy.apply() if called with a function which returned a non-pandas non-scalar object (e.g. a list or numpy array) (GH31441)
- Fixed regression in DataFrame.groupby() whereby taking the minimum or maximum of a column with period dtype would raise a TypeError. (GH31471)
- Fixed regression in DataFrame.groupby() with an empty DataFrame grouping by a level of a MultiIndex (GH31670).
- Fixed regression in DataFrame.apply() with object dtype and non-reducing function (GH31505)
Fixed regression in `to_datetime()` when parsing non-nanosecond resolution datetimes (GH31491)
- Fixed regression in `to_csv()` where specifying an `na_rep` might truncate the values written (GH31447)
- Fixed regression in `to_csv()` where specifying an `na_rep` might truncate the values written (GH31447)
- Fixed regression in `Categorical` construction with `numpy.str` categories (GH31499)
- Fixed regression in `DataFrame.loc()` and `DataFrame.iloc()` when selecting a row containing a single `datetime64` or `timedelta64` column (GH31649)
- Fixed regression where setting `pd.options.display.max_colwidth` was not accepting negative integer. In addition, this behavior has been deprecated in favor of using `None` (GH31532)
- Fixed regression in `objTOJSON.c fix return-type warning` (GH31463)
- Fixed regression in `qcut()` when passed a nullable integer. (GH31389)
- Fixed regression in assigning to a `Series` using a nullable integer dtype (GH31446)
- Fixed performance regression when indexing a `DataFrame` or `Series` with a `MultiIndex` for the index using a list of labels (GH31648)
- Fixed regression in `read_csv()` used in file like object `RawIOBase` is not recognize `encoding` option (GH31575)

**Deprecations**

- Support for negative integer for `pd.options.display.max_colwidth` is deprecated in favor of using `None` (GH31532)

**Bug fixes**

**Datetimelike**

- Fixed bug in `to_datetime()` raising when `cache=True` and out-of-bound values are present (GH31491)

**Numeric**

- Bug in dtypes being lost in `DataFrame.__invert__ (~ operator) with mixed dtypes` (GH31183) and for extension-array backed `Series` and `DataFrame` (GH23087)

**Plotting**

- Plotting tz-aware timeseries no longer gives `UserWarning` (GH31205)

**Interval**

- Bug in `Series.shift()` with `interval` dtype raising a `TypeError` when shifting an interval array of integers or datetimes (GH34195)

**Contributors**

A total of 15 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Daniel Saxton
- Guillaume Lemaitre
- Jeff Reback
- Joris Van den Bossche
5.5.6 What’s new in 1.0.0 (January 29, 2020)

These are the changes in pandas 1.0.0. See Release notes for a full changelog including other versions of pandas.

Note: The pandas 1.0 release removed a lot of functionality that was deprecated in previous releases (see below for an overview). It is recommended to first upgrade to pandas 0.25 and to ensure your code is working without warnings, before upgrading to pandas 1.0.

New deprecation policy

Starting with pandas 1.0.0, pandas will adopt a variant of SemVer to version releases. Briefly,

- Deprecations will be introduced in minor releases (e.g. 1.1.0, 1.2.0, 2.1.0, …)
- Deprecations will be enforced in major releases (e.g. 1.0.0, 2.0.0, 3.0.0, …)
- API-breaking changes will be made only in major releases (except for experimental features)

See Version policy for more.

Enhancements

Using Numba in rolling.apply and expanding.apply

We’ve added an engine keyword to apply() and apply() that allows the user to execute the routine using Numba instead of Cython. Using the Numba engine can yield significant performance gains if the apply function can operate on numpy arrays and the data set is larger (1 million rows or greater). For more details, see rolling apply documentation (GH28987, GH30936)
Defining custom windows for rolling operations

We’ve added a `pandas.api.indexers.BaseIndexer()` class that allows users to define how window bounds are created during rolling operations. Users can define their own `get_window_bounds` method on a `pandas.api.indexers.BaseIndexer()` subclass that will generate the start and end indices used for each window during the rolling aggregation. For more details and example usage, see the `custom window rolling documentation`.

Converting to markdown

We’ve added `to_markdown()` for creating a markdown table (GH11052)

```
In [1]: df = pd.DataFrame({"A": [1, 2, 3], "B": [1, 2, 3]}, index=['a', 'a', 'b'])
In [2]: print(df.to_markdown())
<table>
<thead>
<tr>
<th></th>
<th align="right">A</th>
<th align="right">B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td align="right">1</td>
<td align="right">1</td>
</tr>
<tr>
<td>a</td>
<td align="right">2</td>
<td align="right">2</td>
</tr>
<tr>
<td>b</td>
<td align="right">3</td>
<td align="right">3</td>
</tr>
</tbody>
</table>
```

Experimental new features

Experimental `NA` scalar to denote missing values

A new `pd.NA` value (singleton) is introduced to represent scalar missing values. Up to now, pandas used several values to represent missing data: `np.nan` is used for this for float data, `np.nan` or `None` for object-dtype data and `pd.NaT` for datetime-like data. The goal of `pd.NA` is to provide a “missing” indicator that can be used consistently across data types. `pd.NA` is currently used by the nullable integer and boolean data types and the new string data type (GH28095).

**Warning:** Experimental: the behaviour of `pd.NA` can still change without warning.

For example, creating a Series using the nullable integer dtype:

```
In [3]: s = pd.Series([1, 2, None], dtype="Int64")
In [4]: s
Out[4]:
0  1
1  2
2  <NA>
dtype: Int64
In [5]: s[2]
Out[5]: <NA>
```

Compared to `np.nan`, `pd.NA` behaves differently in certain operations. In addition to arithmetic operations, `pd.NA` also propagates as “missing” or “unknown” in comparison operations:
In [6]: np.nan > 1
Out[6]: False

In [7]: pd.NA > 1
Out[7]: <NA>

For logical operations, pd.NA follows the rules of the three-valued logic (or Kleene logic). For example:

In [8]: pd.NA | True
Out[8]: True

For more, see NA section in the user guide on missing data.

**Dedicated string data type**

We’ve added StringDtype, an extension type dedicated to string data. Previously, strings were typically stored in object-dtype NumPy arrays. (GH29975)

**Warning**: StringDtype is currently considered experimental. The implementation and parts of the API may change without warning.

The 'string' extension type solves several issues with object-dtype NumPy arrays:

1. You can accidentally store a mixture of strings and non-strings in an object dtype array. A StringArray can only store strings.

2. object dtype breaks dtype-specific operations like DataFrame.select_dtypes(). There isn’t a clear way to select just text while excluding non-text, but still object-dtype columns.

3. When reading code, the contents of an object dtype array is less clear than string.

In [9]: pd.Series(['abc', None, 'def'], dtype=pd.StringDtype())
Out[9]:
0   abc
1  <NA>
2   def
dtype: string

You can use the alias "string" as well.

In [10]: s = pd.Series(['abc', None, 'def'], dtype="string")

In [11]: s
Out[11]:
0   abc
1  <NA>
2   def
dtype: string

The usual string accessor methods work. Where appropriate, the return type of the Series or columns of a DataFrame will also have string dtype.
In [12]: s.str.upper()
Out[12]:
0    ABC
1    <NA>
2    DEF
dtype: string

In [13]: s.str.split('b', expand=True).dtypes
Out[13]:
0    string
1    string
dtype: object

String accessor methods returning integers will return a value with `Int64Dtype`

In [14]: s.str.count("a")
Out[14]:
0    1
1    <NA>
2    0
dtype: Int64

We recommend explicitly using the `string` data type when working with strings. See `Text data types` for more.

**Boolean data type with missing values support**

We've added `BooleanDtype / BooleanArray`, an extension type dedicated to boolean data that can hold missing values. The default `bool` data type based on a bool-dtype NumPy array, the column can only hold `True` or `False`, and not missing values. This new `BooleanArray` can store missing values as well by keeping track of this in a separate mask. (GH29555, GH30095, GH31131)

In [15]: pd.Series([True, False, None], dtype=pd.BooleanDtype())
Out[15]:
0    True
1    False
2    <NA>
dtype: boolean

You can use the alias "boolean" as well.

In [16]: s = pd.Series([True, False, None], dtype="boolean")

In [17]: s
Out[17]:
0    True
1    False
2    <NA>
dtype: boolean
Method `convert_dtypes` to ease use of supported extension dtypes

In order to encourage use of the extension dtypes `StringDtype`, `BooleanDtype`, `Int64Dtype`, `Int32Dtype`, etc., that support `pd.NA`, the methods `DataFrame.convert_dtypes()` and `Series.convert_dtypes()` have been introduced. (GH29752) (GH30929)

Example:

```python
In [18]: df = pd.DataFrame({'x': ['abc', None, 'def'],
                        'y': [1, 2, np.nan],
                        'z': [True, False, True]})

In [19]: df
Out[19]:
          x    y    z
0   abc    1  True
1  <NA>    2 False
2   def <NA>  True

In [20]: df.dtypes
Out[20]:
x  object
y  float64
z   bool
dtype: object

In [21]: converted = df.convert_dtypes()

In [22]: converted
Out[22]:
          x   y    z
0   abc    1  True
1 <NA>    2 False
2   def <NA>  True

In [23]: converted.dtypes
Out[23]:
x  string
y   Int64
z  boolean
dtype: object
```

This is especially useful after reading in data using readers such as `read_csv()` and `read_excel()`. See here for a description.
Other enhancements

- `DataFrame.to_string()` added the `max_colwidth` parameter to control when wide columns are truncated (GH9784)
- Added the `na_value` argument to `Series.to_numpy()`, `Index.to_numpy()` and `DataFrame.to_numpy()` to control the value used for missing data (GH30322)
- `MultiIndex.from_product()` infers level names from inputs if not explicitly provided (GH2792)
- `DataFrame.to_latex()` now accepts `caption` and `label` arguments (GH25436)
- DataFrames with `nullable integer`, the `new string dtype` and period data type can now be converted to pyarrow (>=0.15.0), which means that it is supported in writing to the Parquet file format when using the pyarrow engine (GH28368). Full roundtrip to parquet (writing and reading back in with `to_parquet() / read_parquet()`) is supported starting with pyarrow >= 0.16 (GH20612).
- `to_parquet()` now appropriately handles the `schema` argument for user defined schemas in the pyarrow engine. (GH30270)
- `DataFrame.to_json()` now accepts an `indent` integer argument to enable pretty printing of JSON output (GH12004)
- `read_stata()` can read Stata 119 dta files. (GH28250)
- Implemented `pandas.core.window.Window.var()` and `pandas.core.window.Window.std()` functions (GH26597)
- Added encoding argument to `DataFrame.to_string()` for non-ascii text (GH28766)
- Added encoding argument to `DataFrame.to_html()` for non-ascii text (GH28663)
- `Styler.background_gradient()` now accepts `vmin` and `vmax` arguments (GH12145)
- `Styler.format()` added the `na_rep` parameter to help format the missing values (GH21527, GH28358)
- `read_excel()` now can read binary Excel (.xlsb) files by passing `engine='pyxlsb'`. For more details and example usage, see the Binary Excel files documentation. Closes GH8540.
- The `partition_cols` argument in `DataFrame.to_parquet()` now accepts a string (GH27117)
- `pandas.read_json()` now parses NaN, Infinity and -Infinity (GH12213)
- DataFrame constructor preserve `ExtensionArray` dtype with `ExtensionArray` (GH11363)
- `DataFrame.sort_values()` and `Series.sort_values()` have gained `ignore_index` keyword to be able to reset index after sorting (GH30114)
- `DataFrame.sort_index()` and `Series.sort_index()` have gained `ignore_index` keyword to reset index (GH30114)
- `DataFrame.drop_duplicates()` has gained `ignore_index` keyword to reset index (GH30114)
- Added new writer for exporting Stata dta files in versions 118 and 119, `StataWriterUTF8`. These files formats support exporting strings containing Unicode characters. Format 119 supports data sets with more than 32,767 variables (GH23573, GH30959)
- `Series.map()` now accepts `collections.abc.Mapping` subclasses as a mapper (GH29733)
- Added an experimental `attrs` for storing global metadata about a dataset (GH29062)
- `Timestamp.fromisocalendar()` is now compatible with python 3.8 and above (GH28115)
- `DataFrame.to_pickle()` and `read_pickle()` now accept URL (GH30163)
Backwards incompatible API changes

Avoid using names from MultiIndex.levels

As part of a larger refactor to MultiIndex the level names are now stored separately from the levels (GH27242). We recommend using MultiIndex.names to access the names, and Index.set_names() to update the names.

For backwards compatibility, you can still access the names via the levels.

```
In [24]: mi = pd.MultiIndex.from_product([[1, 2], ['a', 'b']], names=['x', 'y'])
In [25]: mi.levels[0].name
Out[25]: 'x'
```

However, it is no longer possible to update the names of the MultiIndex via the level.

```
In [26]: mi.levels[0].name = "new name"
---------------------------------------------------------------------------
RuntimeError                      Traceback (most recent call last)
Input In [26], in <cell line: 1>()
     1 mi.levels[0].name = "new name"
---> 2 mi.levels[0].name = "new name"

File /pandas/pandas/core/indexes/base.py:1661, in Index.name(self, value)
    1657     @name.setter
    1658     def name(self, value: Hashable):
    1659         if self._no_setting_name:
    1660             # Used in MultiIndex.levels to avoid silently ignoring name updates.
    1661             raise RuntimeError(
    1662                 "Cannot set name on a level of a MultiIndex. Use 
    1663                 'MultiIndex.set_names' instead."
    1664         )
    1665     maybe_extract_name(value, None, type(self))
    1666     self._name = value

RuntimeError: Cannot set name on a level of a MultiIndex. Use 'MultiIndex.set_names' instead.
```

```
In [27]: mi.names
Out[27]: FrozenList(['x', 'y'])
```

To update, use MultiIndex.set_names, which returns a new MultiIndex.

```
In [28]: mi2 = mi.set_names("new name", level=0)
In [29]: mi2.names
Out[29]: FrozenList(['new name', 'y'])
```
New repr for `IntervalArray`

`pandas.arrays.IntervalArray` adopts a new `__repr__` in accordance with other array classes (GH25022).

**pandas 0.25.x**

```python
In [1]: pd.arrays.IntervalArray.from_tuples([(0, 1), (2, 3)])
Out[2]:
IntervalArray([(0, 1], (2, 3]),
closed='right',
dtype='interval[int64]')
```

**pandas 1.0.0**

```python
In [30]: pd.arrays.IntervalArray.from_tuples([(0, 1), (2, 3)])
Out[30]:
<IntervalArray>
[(0, 1], (2, 3]]
Length: 2, dtype: interval[int64, right]
```

`DataFrame.rename` now only accepts one positional argument

`DataFrame.rename()` would previously accept positional arguments that would lead to ambiguous or undefined behavior. From pandas 1.0, only the very first argument, which maps labels to their new names along the default axis, is allowed to be passed by position (GH29136).

**pandas 0.25.x**

```python
In [1]: df = pd.DataFrame([[1]])
In [2]: df.rename({0: 1}, {0: 2})
Out[2]:
FutureWarning: ...Use named arguments to resolve ambiguity...
2
  1 1
```

**pandas 1.0.0**

```python
In [3]: df.rename({0: 1}, {0: 2})
Traceback (most recent call last):
...
TypeError: rename() takes from 1 to 2 positional arguments but 3 were given
```

Note that errors will now be raised when conflicting or potentially ambiguous arguments are provided.

**pandas 0.25.x**

```python
In [4]: df.rename({0: 1}, index={0: 2})
Out[4]:
   0
0  1
1  1
In [5]: df.rename(mapper={0: 1}, index={0: 2})
Out[5]:
```

(continues on next page)
pandas 1.0.0

```
In [6]: df.rename({0: 1}, index={0: 2})
Traceback (most recent call last):
...:
TypeError: Cannot specify both 'mapper' and any of 'index' or 'columns'

In [7]: df.rename(mapper={0: 1}, index={0: 2})
Traceback (most recent call last):
...:
TypeError: Cannot specify both 'mapper' and any of 'index' or 'columns'
```

You can still change the axis along which the first positional argument is applied by supplying the `axis` keyword argument.

```
In [31]: df.rename({0: 1})
Out[31]:
   0   1
0   1
1   1

In [32]: df.rename({0: 1}, axis=1)
Out[32]:
   1
0   1
```

If you would like to update both the index and column labels, be sure to use the respective keywords.

```
In [33]: df.rename(index={0: 1}, columns={0: 2})
Out[33]:
   2  1
0   1
```

**Extended verbose info output for DataFrame**

`DataFrame.info()` now shows line numbers for the columns summary (GH17304)

```
pandas 0.25.x

In [1]: df = pd.DataFrame({'int_col': [1, 2, 3],
                          'text_col': ['a', 'b', 'c'],
                          'float_col': [0.0, 0.1, 0.2]})

In [2]: df.info(verbosh=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
int_col 3 non-null int64
text_col 3 non-null object
float_col 3 non-null float64
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

dtypes: float64(1), int64(1), object(1)
memory usage: 152.0+ bytes

pandas 1.0.0

```python
In [34]: df = pd.DataFrame({"int_col": [1, 2, 3],
                        ....: "text_col": ["a", "b", "c"],
                        ....: "float_col": [0.0, 0.1, 0.2]})

In [35]: df.info(verbose=True)
```

```text
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
# Column  Non-Null Count  Dtype
--- ------ -------------- -----  
0 int_col  3 non-null  int64
1 text_col 3 non-null  object
2 float_col 3 non-null  float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

**pandas.array() inference changes**

`pandas.array()` now infers pandas’ new extension types in several cases (GH29791):

1. String data (including missing values) now returns a `arrays.StringArray`.
2. Integer data (including missing values) now returns a `arrays.IntegerArray`.
3. Boolean data (including missing values) now returns the new `arrays.BooleanArray`

pandas 0.25.x

```python
In [1]: pd.array(["a", None])
Out[1]:
<PandasArray>
["a", None]
Length: 2, dtype: object
```

```python
In [2]: pd.array([1, None])
Out[2]:
<PandasArray>
[1, None]
Length: 2, dtype: object
```

pandas 1.0.0

```python
In [36]: pd.array(["a", None])
Out[36]:
<StringArray>
["a", <NA>]
Length: 2, dtype: string
```

(continues on next page)
In [37]: pd.array([1, None])
Out[37]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64

As a reminder, you can specify the dtype to disable all inference.

**arrays.IntegerArray now uses pandas.NA**

arrays.IntegerArray now uses pandas.NA rather than numpy.nan as its missing value marker (GH29964).

pandas 0.25.x

In [1]: a = pd.array([1, 2, None], dtype="Int64")
In [2]: a
Out[2]:
<IntegerArray>
[1, 2, NaN]
Length: 3, dtype: Int64

In [3]: a[2]
Out[3]: nan

pandas 1.0.0

In [38]: a = pd.array([1, 2, None], dtype="Int64")

In [39]: a
Out[39]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64

In [40]: a[2]
Out[40]: <NA>

This has a few API-breaking consequences.

Converting to a NumPy ndarray

When converting to a NumPy array missing values will be pd.NA, which cannot be converted to a float. So calling np.asarray(integer_array, dtype="float") will now raise.

pandas 0.25.x

In [1]: np.asarray(a, dtype="float")
Out[1]:
array([ 1., 2., nan])

pandas 1.0.0
In [41]: np.asarray(a, dtype="float")
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
Input In [41], in <cell line: 1>()
----> 1 np.asarray(a, dtype="float")

File /pandas/pandas/core/arrays/masked.py:415, in BaseMaskedArray.__array__(self, dtype)
     410 def __array__(self, dtype: NpDtype | None = None) -> np.ndarray:
     411     
     412     # the array interface, return my values
     413     # We return an object array here to preserve our scalar values
     414     
--> 415     return self.to_numpy(dtype=dtype)

File /pandas/pandas/core/arrays/masked.py:360, in BaseMaskedArray.to_numpy(self, dtype,␣
    →  copy, na_value)
     354 if self._hasna:
     355     if (  
     356         not is_object_dtype(dtype)  
     357         and not is_string_dtype(dtype)  
     358         and na_value is libmissing.NA  
     359     ):
--> 360         raise ValueError(
     361             f"cannot convert to '{dtype}'-dtype NumPy array "
     362             "with missing values. Specify an appropriate 'na_value' "
     363             "for this dtype."
     364         )
     365     
     366     # don't pass copy to astype --> always need a copy since we are mutating

ValueError: cannot convert to 'float64'-dtype NumPy array with missing values. Specify an appropriate 'na_value' for this dtype.

Use arrays.IntegerArray.to_numpy() with an explicit na_value instead.

In [42]: a.to_numpy(dtype="float", na_value=np.nan)
Out[42]: array([ 1., 2., nan])

Reductions can return `pd.NA``

When performing a reduction such as a sum with skipna=False, the result will now be pd.NA instead of np.nan in presence of missing values (GH30958).

pandas 0.25.x

In [1]: pd.Series(a).sum(skipna=False)
Out[1]:
nan

pandas 1.0.0

In [43]: pd.Series(a).sum(skipna=False)
Out[43]: <NA>

value_counts returns a nullable integer dtype
`Series.value_counts()` with a nullable integer dtype now returns a nullable integer dtype for the values.

**pandas 0.25.x**

```python
In [1]: pd.Series([2, 1, 1, None], dtype="Int64").value_counts().dtype
dtype('int64')
```

**pandas 1.0.0**

```python
In [44]: pd.Series([2, 1, 1, None], dtype="Int64").value_counts().dtype
Int64Dtype()
```

See [Experimental NA scalar to denote missing values](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#missing-values) for more on the differences between `pandas.NA` and `numpy.nan`.

**arrays.IntegerArray comparisons return arrays.BooleanArray**

Comparison operations on a `arrays.IntegerArray` now returns a `arrays.BooleanArray` rather than a NumPy array (GH29964).

**pandas 0.25.x**

```python
In [1]: a = pd.array([1, 2, None], dtype="Int64")
In [2]: a
<IntegerArray>
[1, 2, NaN]
Length: 3, dtype: Int64

In [3]: a > 1
Out[3]:
array([False, True, False])
```

**pandas 1.0.0**

```python
In [45]: a = pd.array([1, 2, None], dtype="Int64")
In [46]: a > 1
Out[46]:
<BooleanArray>
[False, True, <NA>]
Length: 3, dtype: boolean
```

Note that missing values now propagate, rather than always comparing unequal like `numpy.nan`. See [Experimental NA scalar to denote missing values](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#missing-values) for more.
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

**By default Categorical.min() now returns the minimum instead of np.nan**

When `Categorical` contains `np.nan`, `Categorical.min()` no longer return `np.nan` by default (skipna=True) (GH25303)

```
pandas 0.25.x
In [1]: pd.Categorical([1, 2, np.nan], ordered=True).min()
Out[1]: nan

pandas 1.0.0
In [47]: pd.Categorical([1, 2, np.nan], ordered=True).min()
Out[47]: 1
```

**Default dtype of empty pandas.Series**

Initialising an empty `pandas.Series` without specifying a dtype will raise a `DeprecationWarning` now (GH17261). The default dtype will change from `float64` to `object` in future releases so that it is consistent with the behaviour of `DataFrame` and `Index`.

```
pandas 1.0.0
In [1]: pd.Series()
Out[2]:
DeprecationWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.
Series([], dtype: float64)
```

**Result dtype inference changes for resample operations**

The rules for the result dtype in `DataFrame.resample()` aggregations have changed for extension types (GH31359). Previously, pandas would attempt to convert the result back to the original dtype, falling back to the usual inference rules if that was not possible. Now, pandas will only return a result of the original dtype if the scalar values in the result are instances of the extension dtype’s scalar type.

```
In [48]: df = pd.DataFrame({'A': ['a', 'b']}, dtype='category',
                   index=pd.date_range('2000', periods=2))

In [49]: df
Out[49]:
         A
2000-01-01 a
2000-01-02 b

pandas 0.25.x
In [1]> df.resample("2D").agg(lambda x: 'a').A.dtype
Out[1]:
CategoricalDtype(categories=['a', 'b'], ordered=False)
```

```
pandas 0.25.x
In [1]> df.resample("2D").agg(lambda x: 'a').A.dtype
Out[1]:
CategoricalDtype(categories=['a', 'b'], ordered=False)
```
This fixes an inconsistency between `resample` and `groupby`. This also fixes a potential bug, where the values of the result might change depending on how the results are cast back to the original dtype.

**pandas 0.25.x**

```
In [1]: df.resample("2D").agg(lambda x: 'c')
Out[1]:
   A
0  NaN
```

**pandas 1.0.0**

```
In [51]: df.resample("2D").agg(lambda x: 'c')
Out[51]:
   A
2000-01-01  c
```

## Increased minimum version for Python

pandas 1.0.0 supports Python 3.6.1 and higher (GH29212).

## Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH29766, GH29723). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.13.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>pytz</td>
<td>2015.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.6.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.6.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>4.0.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.


## Build changes

pandas has added a `pyproject.toml` file and will no longer include cythonized files in the source distribution uploaded to PyPI (GH28341, GH20775). If you’re installing a built distribution (wheel) or via conda, this shouldn’t have any effect on you. If you’re building pandas from source, you should no longer need to install Cython into your build environment before calling `pip install pandas`.

## Other API changes

- `core.groupby.GroupBy.transform` now raises on invalid operation names (GH27489)
- `pandas.api.types.infer_dtype()` will now return “integer-na” for integer and `np.nan` mix (GH27283)
- `MultiIndex.from_arrays()` will no longer infer names from arrays if `names=None` is explicitly provided (GH27292)
- In order to improve tab-completion, pandas does not include most deprecated attributes when introspecting a pandas object using `dir` (e.g. `dir(df)`). To see which attributes are excluded, see an object’s `_deprecations` attribute, for example `pd.DataFrame._deprecations` (GH28805).
- The returned dtype of `unique()` now matches the input dtype. (GH27874)
- Changed the default configuration value for `options.matplotlib.register_converters` from `True` to "auto" (GH18720). Now, pandas custom formatters will only be applied to plots created by pandas, through `plot()`. Previously, pandas’ formatters would be applied to all plots created after a `plot()`. See `units registration` for more.
- `Series.dropna()` has dropped its `**kwargs` argument in favor of a single `how` parameter. Supplying anything else than how to `**kwargs` raised a `TypeError` previously (GH29388)
- When testing pandas, the new minimum required version of pytest is 5.0.1 (GH29664)
- `Series.str.__iter__()` was deprecated and will be removed in future releases (GH28277).
• Added <NA> to the list of default NA values for `read_csv()` (GH30821)

Documentation improvements

• Added new section on *Scaling to large datasets* (GH28315).
• Added sub-section on *Query MultiIndex* for HDF5 datasets (GH28791).

Deprecations

• `Series.item()` and `Index.item()` have been _undeprecated_ (GH29250)
• `Index.set_value` has been deprecated. For a given index idx, array arr, value in idx of idx_val and a new value of val, `idx.set_value(arr, idx_val, val)` is equivalent to `arr[idx.get_loc(idx_val)] = val`, which should be used instead (GH28621).
• `is_extension_type()` is deprecated, `is_extension_array_dtype()` should be used instead (GH29457)
• `eval()` keyword argument “trueiv” is deprecated and will be removed in a future version (GH29812)
• `DateOffset.isAnchored()` and `DateOffset.onOffset()` are deprecated and will be removed in a future version, use `DateOffset.is_anchored()` and `DateOffset.is_on_offset()` instead (GH30340)
• `pandas.tseries.frequencies.get_offset` is deprecated and will be removed in a future version, use `pandas.tseries.frequencies.to_offset` instead (GH4205)
• `Categorical.take_nd()` and `CategoricalIndex.take_nd()` are deprecated, use `Categorical.take()` and `CategoricalIndex.take()` instead (GH27745)
• The parameter `numeric_only` of `Categorical.min()` and `Categorical.max()` is deprecated and replaced with `skipna` (GH25303)
• The parameter `label` in `lreshape()` has been deprecated and will be removed in a future version (GH29742)
• `pandas.core.index` has been deprecated and will be removed in a future version, the public classes are available in the top-level namespace (GH19711)
• `pandas.json_normalize()` is now exposed in the top-level namespace. Usage of `json_normalize` as `pandas.io.json.json_normalize` is now deprecated and it is recommended to use `json_normalize` as `pandas.json_normalize()` instead (GH27586).
• The numpy argument of `pandas.read_json()` is deprecated (GH28512).
• `DataFrame.to_stata()`, `DataFrame.to_feather()`, and `DataFrame.to_parquet()` argument “fname” is deprecated, use “path” instead (GH23574)
• The deprecated internal attributes `_start`, `_stop` and `_step` of `RangeIndex` now raise a `FutureWarning` instead of a `DeprecationWarning` (GH26581)
• The `pandas.util.testing` module has been deprecated. Use the public API in `pandas.testing` documented at `Testing functions` (GH16232).
• `pandas.SparseArray` has been deprecated. Use `pandas.arrays.SparseArray (arrays.SparseArray)` instead. (GH30642)
• The parameter `is_copy` of `Series.take()` and `DataFrame.take()` has been deprecated and will be removed in a future version. (GH27357)
• Support for multi-dimensional indexing (e.g. `index[:, None]`) on a `Index` is deprecated and will be removed in a future version, convert to a numpy array before indexing instead (GH30588)
• The `pandas.np` submodule is now deprecated. Import numpy directly instead (GH30296)
The `pandas.datetime` class is now deprecated. Import from `datetime` instead (GH30610)

`diff` will raise a `TypeError` rather than implicitly losing the dtype of extension types in the future. Convert to the correct dtype before calling `diff` instead (GH31025)

Selecting Columns from a Grouped DataFrame

When selecting columns from a `DataFrameGroupBy` object, passing individual keys (or a tuple of keys) inside single brackets is deprecated, a list of items should be used instead. (GH23566) For example:

```python
def = pd.DataFrame({
    "A": ["foo", "bar", "foo", "bar", "foo", "foo"],
    "B": np.random.randn(8),
    "C": np.random.randn(8),
})
g = df.groupby('A')

# single key, returns SeriesGroupBy
  g['B']

# tuple of single key, returns SeriesGroupBy
  g[['B',]]

# tuple of multiple keys, returns DataFrameGroupBy, raises FutureWarning
  g[['B', 'C']]  

# multiple keys passed directly, returns DataFrameGroupBy, raises FutureWarning
# (implicitly converts the passed strings into a single tuple)
  g['B', 'C']

# proper way, returns DataFrameGroupBy
  g[['B', 'C']]
```

Removal of prior version deprecations/changes

Removed SparseSeries and SparseDataFrame

SparseSeries, SparseDataFrame and the `DataFrame.to_sparse` method have been removed (GH28425). We recommend using a `Series` or `DataFrame` with sparse values instead. See Migrating for help with migrating existing code.

Matplotlib unit registration

Previously, pandas would register converters with matplotlib as a side effect of importing pandas (GH18720). This changed the output of plots made via matplotlib plots after pandas was imported, even if you were using matplotlib directly rather than `plot()`.

To use pandas formatters with a matplotlib plot, specify

```python
In [1]: import pandas as pd
In [2]: pd.options.plotting.matplotlib.register_converters = True
```

Note that plots created by `DataFrame.plot()` and `Series.plot()` do register the converters automatically. The only behavior change is when plotting a date-like object via `matplotlib.pyplot.plot` or `matplotlib.Axes.plot`. See Custom formatters for timeseries plots for more.

Other removals
• Removed the previously deprecated keyword “index” from `read_stata()`, `StataReader`, and `StataReader.read()`, use “index_col” instead (GH17328)
• Removed `StataReader.data` method, use `StataReader.read()` instead (GH9493)
• Removed `pandas.plotting._matplotlib.tsplot` use `Series.plot()` instead (GH19980)

• `pandas.tseries.converter.register` has been moved to `pandas.plotting.register_matplotlib_converters()` (GH18307)
• `Series.plot()` no longer accepts positional arguments, pass keyword arguments instead (GH30003)
• `DataFrame.hist()` and `Series.hist()` no longer allows `figsize="default"`, specify figure size by passing a tuple instead (GH30003)
• Floordiv of integer-dtyped array by `Timedelta` now raises `TypeError` (GH21036)

• `TimedeltaIndex` and `DatetimeIndex` no longer accept non-nanosecond dtype strings like “timedelta64” or “datetime64”, use “timedelta64[ns]” and “datetime64[ns]” instead (GH24806)
• Changed the default “skipna” argument in `pandas.api.types.infer_dtype()` from False to True (GH24050)
• Removed `Series.ix` and `DataFrame.ix` (GH26438)
• Removed `Index.summary` (GH18217)
• Removed the previously deprecated keyword “fastpath” from the `Index` constructor (GH23110)
• Removed `Series.get_value`, `Series.set_value`, `DataFrame.get_value`, `DataFrame.set_value` (GH17739)
• Removed `Series.compound` and `DataFrame.compound` (GH26405)
• Changed the default “inplace” argument in `DataFrame.set_index()` and `Series.set_axis()` from None to False (GH27600)
• Removed `Series.cat.categorical`, `Series.cat.index`, `Series.cat.name` (GH24751)
• Removed the previously deprecated keyword “box” from `to_datetime()` and `to_timedelta()`; in addition these now always returns `DatetimeIndex`, `TimedeltaIndex`, `Index`, `Series`, or `DataFrame` (GH24486)
• `to_timedelta()`, `Timedelta`, and `TimedeltaIndex` no longer allow “M”, “y”, or “Y” for the “unit” argument (GH23264)
• Removed the previously deprecated keyword “time_rule” from (non-public) `offsets.generate_range`, which has been moved to `core.arrays._ranges.generate_range()` (GH24157)

• `DataFrame.loc()` or `Series.loc()` with listlike indexers and missing labels will no longer reindex (GH17295)
• `DataFrame.to_excel()` and `Series.to_excel()` with non-existent columns will no longer reindex (GH17295)
• Removed the previously deprecated keyword “join_axes” from `concat()`; use `reindex_like` on the result instead (GH22318)
• Removed the previously deprecated keyword “by” from `DataFrame.sort_index()`, use `DataFrame.sort_values()` instead (GH10726)
• Removed support for nested renaming in `DataFrame.aggregate()`, `Series.aggregate()`, `core.groupby.DataFrameGroupBy.aggregate()`, `core.groupby.SeriesGroupBy.aggregate()`, `core.window.rolling.Rolling.aggregate()` (GH18529)
• Passing datetime64 data to `TimedeltaIndex` or `timedelta64` data to `DatetimeIndex` now raises `TypeError` (GH23539, GH23937)
• Passing \texttt{int64} values to \texttt{DatetimeIndex} and a timezone now interprets the values as nanosecond timestamps in UTC, not wall times in the given timezone (GH24559)

• A tuple passed to \texttt{DataFrame.groupby()} is now exclusively treated as a single key (GH18314)

• Removed \texttt{Index.contains}, use \texttt{key in index} instead (GH30103)

• Addition and subtraction of \texttt{int} or integer-arrays is no longer allowed in \texttt{Timestamp, DatetimeIndex, TimedeltaIndex}, use \texttt{obj + n * obj.freq} instead of \texttt{obj + n} (GH22535)

• Removed \texttt{Series.ptp} (GH21614)

• Removed \texttt{Series.from_array} (GH18258)

• Removed \texttt{DataFrame.from_items} (GH18458)

• Removed \texttt{DataFrame.as_matrix, Series.as_matrix} (GH18458)

• Removed \texttt{Series.asobject} (GH18477)

• Removed \texttt{DataFrame.as_blocks, Series.as_blocks, DataFrame.blocks, Series.blocks} (GH17656)

• \texttt{pandas.Series.str.cat()} now defaults to aligning \texttt{others}, using \texttt{join='left'} (GH27611)

• \texttt{pandas.Series.str.cat()} does not accept list-likes \texttt{within} list-likes anymore (GH27611)

• \texttt{Series.where()} with \texttt{Categorical} dtype (or \texttt{DataFrame.where()} with \texttt{Categorical} column) no longer allows setting new categories (GH24114)

• Removed the previously deprecated keywords “start”, “end”, and “periods” from the \texttt{DatetimeIndex, TimedeltaIndex, and PeriodIndex} constructors; use \texttt{date_range()}, \texttt{timedelta_range()}, and \texttt{period_range()} instead (GH23919)

• Removed the previously deprecated keyword “verify_integrity” from the \texttt{DatetimeIndex} and \texttt{TimedeltaIndex} constructors (GH23919)

• Removed the previously deprecated keyword “fastpath” from \texttt{pandas.core.internals.blocks.make_block} (GH19265)

• Removed the previously deprecated keyword “dtype” from \texttt{Block.make_block_same_class()} (GH19434)

• Removed \texttt{ExtensionArray._formatting_values}. Use \texttt{ExtensionArray._formatter} instead. (GH23601)

• Removed \texttt{MultiIndex.to_hierarchical} (GH21613)

• Removed \texttt{MultiIndex.labels}, use \texttt{MultiIndex.codes} instead (GH23752)

• Removed the previously deprecated keyword “labels” from the \texttt{MultiIndex} constructor, use “codes” instead (GH23752)

• Removed \texttt{MultiIndex.set_labels}, use \texttt{MultiIndex.set_codes()} instead (GH23752)

• Removed the previously deprecated keyword “labels” from \texttt{MultiIndex.set_codes()}, \texttt{MultiIndex.copy()}, \texttt{MultiIndex.drop()}, use “codes” instead (GH23752)

• Removed support for legacy HDF5 formats (GH29787)

• Passing a dtype alias (e.g. ‘datetime64[ns, UTC]’) to \texttt{DatetimeTZDtype} is no longer allowed, use \texttt{DatetimeTZDtype.construct_from_string()} instead (GH23990)

• Removed the previously deprecated keyword “skip_footer” from \texttt{read_excel()}; use “skipfooter” instead (GH18836)

• \texttt{read_excel()} no longer allows an integer value for the parameter \texttt{usecols}, instead pass a list of integers from 0 to \texttt{usecols inclusive} (GH23635)
• Removed the previously deprecated keyword “convert_datetime64” from `DataFrame.to_records()` (GH18902)

• Removed `IntervalIndex.from_intervals` in favor of the `IntervalIndex` constructor (GH19263)

• Changed the default “keep_tz” argument in `DatetimeIndex.to_series()` from `None` to `True` (GH23739)

• Removed `api.types.is_period` and `api.types.is_datetimetz` (GH23917)

• Ability to read pickles containing `Categorical` instances created with pre-0.16 version of pandas has been removed (GH27538)

• Removed `pandas.tseries.plotting.tsplot` (GH18627)

• Removed the previously deprecated keywords “reduce” and “broadcast” from `DataFrame.apply()` (GH18577)

• Removed the previously deprecated `assert_raises_regex` function in `pandas._testing` (GH29174)

• Removed the previously deprecated `FrozenNDArray` class in `pandas.core.indexes.frozen` (GH29335)

• Removed the previously deprecated keyword “nthreads” from `read_feather()`, use “use_threads” instead (GH23053)

• Removed `Index.is_lexsorted_for_tuple` (GH29305)

• Removed support for nested renaming in `DataFrame.aggregate()`, `Series.aggregate()`, `core.groupby.DataFrameGroupBy.aggregate()`, `core.groupby.SeriesGroupBy.aggregate()`, `core.window.Rolling.Rolling.aggregate()` (GH29608)

• Removed `Series.valid`; use `Series.dropna()` instead (GH18800)

• Removed `DataFrame.is_copy`, `Series.is_copy` (GH18812)

• Removed `DataFrame.get_ftype_counts`, `Series.get_ftype_counts` (GH18243)

• Removed `DataFrame.ftypes`, `Series.ftypes`, `Series.ftype` (GH26744)

• Removed `Index.get_duplicates`, use `idx[idx.duplicated()].unique()` instead (GH20239)

• Removed `Series.clip_upper`, `Series.clip_lower`, `DataFrame.clip_upper`, `DataFrame.clip_lower` (GH24203)

• Removed the ability to alter `DatetimeIndex.freq`, `TimedeltaIndex.freq`, or `PeriodIndex.freq` (GH20772)

• Removed `DatetimeIndex.offset` (GH20730)

• Removed `DatetimeIndex.asobject`, `TimedeltaIndex.asobject`, `PeriodIndex.asobject`, use `astype(object)` instead (GH29801)

• Removed the previously deprecated keyword “order” from `factorize()` (GH19751)

• Removed the previously deprecated keyword “encoding” from `read_stata()` and `DataFrame.to_stata()` (GH21400)

• Changed the default “sort” argument in `concat()` from `None` to `False` (GH20613)

• Removed the previously deprecated keyword “raise_conflict” from `DataFrame.update()`, use “errors” instead (GH23585)

• Removed the previously deprecated keyword “n” from `DatetimeIndex.shift()`, `TimedeltaIndex.shift()`, `PeriodIndex.shift()`, use “periods” instead (GH22458)

• Removed the previously deprecated keywords “how”, “fill_method”, and “limit” from `DataFrame.resample()` (GH30139)
• Passing an integer to `Series.fillna()` or `Dataframe.fillna()` with `timedelta64[ns]` dtype now raises `TypeError` (GH24694)
• Passing multiple axes to `DataFrame.dropna()` is no longer supported (GH20995)
• Removed `Series.nonzero`, use `to_numpy().nonzero()` instead (GH24048)
• Passing floating dtype codes to `Categorical.from_codes()` is no longer supported, pass `codes.astype(np.int64)` instead (GH21775)
• Removed the previously deprecated keyword “pat” from `Series.str.partition()` and `Series.str.rpartition()`, use “sep” instead (GH23767)
• Removed `Series.put` (GH27106)
• Removed `Series.real, Series.imag` (GH27106)
• Removed `Series.to_dense, DataFrame.to_dense` (GH26684)
• Removed `Index.dtype_str, use str(index.dtype)` instead (GH27106)
• `Categorical.ravel()` returns a `Categorical` instead of a `ndarray` (GH27199)
• The ‘outer’ method on Numpy ufuncs, e.g. `np.subtract.outer` operating on `Series` objects is no longer supported, and will raise `NotImplementedError` (GH27198)
• Removed `Series.get_dtype_counts` and `DataFrame.get_dtype_counts` (GH27145)
• Changed the default “fill_value” argument in `Categorical.take()` from `True` to `False` (GH20841)
• Changed the default value for the `raw` argument in `Series.rolling().apply().DataFrame.rolling().apply().Series.expanding().apply().and DataFrame.expanding().apply()` from `None` to `False` (GH20584)
• Removed deprecated behavior of `Series.argmin()` and `Series.argmax()`, use `Series.idxmin()` and `Series.idxmax()` for the old behavior (GH16955)
• Passing a tz-aware datetime.datetime or `Timestamp` into the `Timestamp` constructor with the `tz` argument now raises a `ValueError` (GH23621)
• Removed `Series.base, Index.base, Categorical.base, Series.flags, Index.flags, PeriodArray.flags, Series.strides, Index.strides, Series.itemsize, Index.itemsize, Series.data, Index.data` (GH20721)
• Changed `Timedelta.resolution()` to match the behavior of the standard library `datetime.timedelta.resolution`, for the old behavior, use `Timedelta.resolution_string()` (GH26839)
• Removed `Timestamp.weekday_name, DatetimeIndex.weekday_name, and Series.dt.weekday_name` (GH18164)
• Removed the previously deprecated keyword “errors” in `Timestamp.tz_localize(), DatetimeIndex.tz_localize(), and Series.tz_localize()` (GH22644)
• Changed the default “ordered” argument in `CategoricalDtype` from `None` to `False` (GH26336)
• `Series.set_axis()` and `DataFrame.set_axis()` now require “labels” as the first argument and “axis” as an optional named parameter (GH30089)
• Removed `to_msgpack, read_msgpack, DataFrame.to_msgpack, Series.to_msgpack` (GH27103)
• Removed `Series.compress` (GH21930)
• Removed the previously deprecated keyword “fill_value” from `Categorical.fillna()`, use “value” instead (GH19269)
• Removed the previously deprecated keyword “data” from `andrews_curves()`, use “frame” instead (GH6956)
• Removed the previously deprecated keyword “data” from parallel_coordinates(), use “frame” instead (GH6956)
• Removed the previously deprecated keyword “colors” from parallel_coordinates(), use “color” instead (GH6956)
• Removed the previously deprecated keywords “verbose” and “private_key” from read_gbq() (GH30200)
• Calling np.array and np.asarray on tz-aware Series and DatetimeIndex will now return an object array of tz-aware Timestamp (GH24596)

Performance improvements

• Performance improvement in DataFrame arithmetic and comparison operations with scalars (GH24990, GH29853)
• Performance improvement in indexing with a non-unique IntervalIndex (GH27489)
• Performance improvement in MultiIndex.is_monotonic (GH27495)
• Performance improvement in cut() when bins is an IntervalIndex (GH27668)
• Performance improvement when initializing a DataFrame using a range (GH30171)
• Performance improvement in DataFrame.corr() when method is "spearman" (GH28099)
• Performance improvement in DataFrame.replace() when provided a list of values to replace (GH28317)
• Performance improvement in DataFrame.select_dtypes() by using vectorization instead of iterating over a loop (GH28795)
• Performance improvement when comparing a Categorical with a scalar and the scalar is not found in the categories (GH29820)
• Performance improvement in Index.equals() and MultiIndex.equals() (GH29134)
• Performance improvement in infer_dtype() when skipna is True (GH28814)

Bug fixes

Categorical

• Added test to assert the fillna() raises the correct ValueError message when the value isn’t a value from categories (GH13628)
• Bug in Categorical.astype() where NaN values were handled incorrectly when casting to int (GH28406)
• DataFrame.reindex() with a CategoricalIndex would fail when the targets contained duplicates, and wouldn’t fail if the source contained duplicates (GH28107)
• Bug in Categorical.astype() not allowing for casting to extension dtypes (GH28668)
• Bug where merge() was unable to join on categorical and extension dtype columns (GH28668)
• Categorical.searchsorted() and CategoricalIndex.searchsorted() now work on unordered categories also (GH21667)

• Added test to assert roundtripping to parquet with DataFrame.to_parquet() or read_parquet() will preserve Categorical dtypes for string types (GH27955)

• Changed the error message in Categorical.remove_categories() to always show the invalid removals as a set (GH28669)

• Using date accessors on a categorical dtyped Series of datetimes was not returning an object of the same type as if one used the str() / dt() on a Series of that type. E.g. when accessing Series.dt.tz_localize() on a Categorical with duplicate entries, the accessor was skipping duplicates (GH27952)

• Bug in DataFrame.replace() and Series.replace() that would give incorrect results on categorical data (GH26988)

• Bug where calling Categorical.min() or Categorical.max() on an empty Categorical would raise a numpy exception (GH30227)

• The following methods now also correctly output values for unobserved categories when called through groupby(..., observed=False) * core.groupby.SeriesGroupBy.count() * core.groupby.SeriesGroupBy.size() * core.groupby.SeriesGroupBy.nunique() * core.groupby.SeriesGroupBy.nth()

Datetimelike

• Bug in Series.__setitem__() incorrectly casting np.timedelta64("NaT") to np.datetime64("NaT") when inserting into a Series with datetime64 dtype (GH27311)

• Bug in Series.dt() property lookups when the underlying data is read-only (GH27529)

• Bug in HDFStore.__getitem__ incorrectly reading tz attribute created in Python 2 (GH26443)

• Bug in to_datetime() where passing arrays of malformed str with errors="coerce" could incorrectly lead to raising ValueError (GH28299)

• Bug in core.groupby.SeriesGroupBy.nunique() where NaT values were interfering with the count of unique values (GH27951)

• Bug in Timestamp subtraction when subtracting a Timestamp from a np.datetime64 object incorrectly raising TypeError (GH28286)

• Addition and subtraction of integer or integer-dtype arrays with Timestamp will now raise NullFrequencyError instead of ValueError (GH28268)

• Bug in Series and DataFrame with integer dtype failing to raise TypeError when adding or subtracting a np.datetime64 object (GH28080)

• Bug in Series.astype(), Index.astype(), and DataFrame.astype() failing to handle NaT when casting to an integer dtype (GH28492)

• Bug in Week with weekday incorrectly raising AttributeError instead of TypeError when adding or subtracting an invalid type (GH28530)

• Bug in DataFrame arithmetic operations when operating with a Series with dtype 'timedelta64[ns]' (GH28049)

• Bug in core.groupby.generic.SeriesGroupBy.apply() raising ValueError when a column in the original DataFrame is a datetime and the column labels are not standard integers (GH28247)

• Bug in pandas._config.localization.get_locales() where the locales -a encodes the locales list as windows-1252 (GH23638, GH24760, GH27368)
• Bug in `Series.var()` failing to raise `TypeError` when called with `timedelta64[ns]` dtype (GH28289)

• Bug in `DatetimeIndex.strftime()` and `Series.dt.strftime()` where NaT was converted to the string 'NaT' instead of `np.nan` (GH29578)

• Bug in masking datetime-like arrays with a boolean mask of an incorrect length not raising an `IndexError` (GH30308)

• Bug in `Timestamp.resolution` being a property instead of a class attribute (GH29910)

• Bug in `pandas.to_datetime()` when called with `None` raising `TypeError` instead of returning `NaT` (GH30011)

• Bug in `pandas.to_datetime()` failing for deques when using `cache=True` (the default) (GH29403)

• Bug in `Series.item()` with `datetime64` or `timedelta64` dtype, `DatetimeIndex.item()`, and `TimedeltaIndex.item()` returning an integer instead of a `Timestamp` or `Timedelta` (GH30175)

• Bug in `DatetimeIndex` addition when adding a non-optimized `DateOffset` incorrectly dropping timezone information (GH30336)

• Bug in `DataFrame.drop()` where attempting to drop non-existent values from a DatetimeIndex would yield a confusing error message (GH30399)

• Bug in `DataFrame.append()` would remove the timezone-awareness of new data (GH30238)

• Bug in `Series.cummin()` and `Series.cummax()` with timezone-aware dtype incorrectly dropping its timezone (GH15553)

• Bug in `DatetimeArray`, `TimedeltaArray`, and `PeriodArray` where inplace addition and subtraction did not actually operate inplace (GH24115)

• Bug in `pandas.to_datetime()` when called with `Series` storing `IntegerArray` raising `TypeError` instead of returning `Series` (GH30050)

• Bug in `date_range()` with custom business hours as `freq` and given number of `periods` (GH30593)

• Bug in `PeriodIndex` comparisons with incorrectly casting integers to `Period` objects, inconsistent with the `Period` comparison behavior (GH30722)

• Bug in `DatetimeIndex.insert()` raising a `ValueError` instead of a `TypeError` when trying to insert a timezone-aware `Timestamp` into a timezone-naive `DatetimeIndex`, or vice-versa (GH30806)

**Timedelta**

• Bug in subtracting a `TimedeltaIndex` or `TimedeltaArray` from a `np.datetime64` object (GH29558)

**Timezones**
Numeric

- Bug in `DataFrame.quantile()` with zero-column `DataFrame` incorrectly raising (GH23925)
- `DataFrame` flex inequality comparisons methods (`DataFrame.lt()`, `DataFrame.le()`, `DataFrame.gt()`, `DataFrame.ge()`) with object-dtype and complex entries failing to raise `TypeError` like their `Series` counterparts (GH28079)
- Bug in `DataFrame` logical operations (&, |, ^) not matching `Series` behavior by filling NA values (GH28741)
- Bug in `DataFrame.interpolate()` where specifying axis by name references variable before it is assigned (GH29142)
- Bug in `Series.var()` not computing the right value with a nullable integer dtype series not passing through `ddof` argument (GH29128)
- Improved error message when using `frac > 1` and `replace = False` (GH27451)
- Bug in numeric indexes resulted in it being possible to instantiate an `Int64Index`, `UInt64Index`, or `Float64Index` with an invalid dtype (e.g. datetime-like) (GH29539)
- Bug in `UInt64Index` precision loss while constructing from a list with values in the `np.uint64` range (GH29526)
- Bug in `NumericIndex` construction that caused indexing to fail when integers in the `np.uint64` range were used (GH28023)
- Bug in `NumericIndex` construction that caused `UInt64Index` to be casted to `Float64Index` when integers in the `np.uint64` range were used to index a `DataFrame` (GH28279)
- Bug in `Series.interpolate()` when using method=`index` with an unsorted index, would previously return incorrect results. (GH21037)
- Bug in `DataFrame.round()` where a `DataFrame` with a `CategoricalIndex` of `IntervalIndex` columns would incorrectly raise a `TypeError` (GH30063)
- Bug in `Series.pct_change()` and `DataFrame.pct_change()` when there are duplicated indices (GH30463)
- Bug in `DataFrame` cumulative operations (e.g. cumsum, cummax) incorrect casting to object-dtype (GH19296)
- Bug in `diff` losing the dtype for extension types (GH30889)
- Bug in `DataFrame.diff` raising an `IndexError` when one of the columns was a nullable integer dtype (GH30967)

Conversion

- 

Strings

- Calling `Series.str.isalnum()` (and other “ismethods”) on an empty `Series` would return an object dtype instead of `bool` (GH29624)
- 

Interval

- Bug in `IntervalIndex.get_indexer()` where a `Categorical` or `CategoricalIndex` target would incorrectly raise a `TypeError` (GH30063)
- Bug in `pandas.core.dtypes.cast.infer_dtype_from_scalar` where passing `pandas_dtype=True` did not infer `IntervalDtype` (GH30337)
- Bug in `Series` constructor where constructing a `Series` from a list of `Interval` objects resulted in object dtype instead of `IntervalDtype` (GH23563)
- Bug in `IntervalDtype` where the `kind` attribute was incorrectly set as `None` instead of "O" (GH30568)
- Bug in `IntervalIndex`, `IntervalArray`, and `Series` with interval data where equality comparisons were incorrect (GH24112)

Indexing

- Bug in assignment using a reverse slicer (GH26939)
- Bug in `DataFrame.explode()` would duplicate frame in the presence of duplicates in the index (GH28010)
- Bug in reindexing a `PeriodIndex()` with another type of index that contained a `Period` (GH28323, GH28337)
- Fix assignment of column via `.loc` with numpy non-ns datetime type (GH27395)
- Bug in `Float64Index.astype()` where `np.inf` was not handled properly when casting to an integer dtype (GH28475)
- `Index.union()` could fail when the left contained duplicates (GH28257)
- Bug when indexing with `.loc` where the index was a `CategoricalIndex` with non-string categories didn’t work (GH17569, GH30225)
- `Index.get_indexer_non_unique()` could fail with `TypeError` in some cases, such as when searching for ints in a string index (GH28257)
- Bug in `Float64Index.get_loc()` incorrectly raising `TypeError` instead of `KeyError` (GH29189)
- Bug in `DataFrame.loc()` with incorrect dtype when setting Categorical value in 1-row DataFrame (GH25495)
- `MultiIndex.get_loc()` can’t find missing values when input includes missing values (GH19132)
- Bug in `Series.__setitem__()` incorrectly assigning values with boolean indexer when the length of new data matches the number of True values and new data is not a `Series` or an `np.array` (GH30567)
- Bug in indexing with a `PeriodIndex` incorrectly accepting integers representing years, use e.g. `ser.loc["2007"]` instead of `ser.loc[2007]` (GH30763)

Missing

-
**MultiIndex**

- Constructor for `MultiIndex` verifies that the given `sortorder` is compatible with the actual `lexsort_depth` if `verify_integrity` parameter is `True` (the default) (GH28735)

- Series and MultiIndex `.drop` with `MultiIndex` raise exception if labels not in given in level (GH8594)

**IO**

- `read_csv()` now accepts binary mode file buffers when using the Python csv engine (GH23779)

- Bug in `DataFrame.to_json()` where using a Tuple as a column or index value and using `orient="columns"` or `orient="index"` would produce invalid JSON (GH20500)

- Improve infinity parsing. `read_csv()` now interprets `Infinity`, `-Infinity` as floating point values (GH10065)

- Bug in `DataFrame.to_csv()` where values were truncated when the length of `na_rep` was shorter than the text input data. (GH25099)

- Bug in `DataFrame.to_string()` where values were truncated using display options instead of outputting the full content (GH9784)

- Bug in `DataFrame.to_json()` where a datetime column label would not be written out in ISO format with `orient="table"` (GH28130)

- Bug in `DataFrame.to_parquet()` where writing to GCS would fail with `engine='fastparquet'` if the file did not already exist (GH28326)

- Bug in `read_hdf()` closing stores that it didn’t open when Exceptions are raised (GH28699)

- Bug in `DataFrame.read_json()` where using `orient="index"` would not maintain the order (GH28557)

- Bug in `DataFrame.to_html()` where the length of the `formatters` argument was not verified (GH28469)

- Bug in `DataFrame.read_excel()` with `engine='ods'` when `sheet_name` argument references a non-existent sheet (GH27676)

- Bug in `pandas.io.formats.style.Styler()` formatting for floating values not displaying decimals correctly (GH13257)

- Bug in `DataFrame.to_html()` when using `formatters=<list>` and `max_cols` together. (GH25955)

- Bug in `Styler.background_gradient()` not able to work with dtype `Int64` (GH28869)

- Bug in `DataFrame.to_clipboard()` which did not work reliably in ipython (GH22707)

- Bug in `read_json()` where default encoding was not set to utf-8 (GH29565)

- Bug in `PythonParser` where str and bytes were being mixed when dealing with the decimal field (GH29650)

- `read_gbq()` now accepts `progress_bar_type` to display progress bar while the data downloads. (GH29857)

- Bug in `pandas.io.json.json_normalize()` where a missing value in the location specified by `record_path` would raise a `TypeError` (GH30148)

- `read_excel()` now accepts binary data (GH15914)

- Bug in `read_csv()` in which encoding handling was limited to just the string utf-16 for the C engine (GH24130)
Plotting

- Bug in `Series.plot()` not able to plot boolean values (GH23719)
- Bug in `DataFrame.plot()` not able to plot when no rows (GH27758)
- Bug in `DataFrame.plot()` producing incorrect legend markers when plotting multiple series on the same axis (GH18222)
- Bug in `DataFrame.plot()` when `kind='box'` and data contains datetime or timedelta data. These types are now automatically dropped (GH22799)
- Bug in `DataFrame.plot.line()` and `DataFrame.plot.area()` produce wrong xlim in x-axis (GH27686, GH25160, GH24784)
- Bug where `DataFrame.boxplot()` would not accept a `color` parameter like `DataFrame.plot.box()` (GH26214)
- Bug in the `xticks` argument being ignored for `DataFrame.plot.bar()` (GH14119)
- `set_option()` now validates that the plot backend provided to 'plotting.backend' implements the backend when the option is set, rather than when a plot is created (GH28163)
- `DataFrame.plot()` now allow a `backend` keyword argument to allow changing between backends in one session (GH28619).
- Bug in color validation incorrectly raising for non-color styles (GH29122).
- Allow `DataFrame.plot.scatter()` to plot objects and datetime type data (GH18755, GH30391)
- Bug in `DataFrame.hist()`, `xrot=0` does not work with by and subplots (GH30288).

GroupBy/resample/rolling

- Bug in `core.groupby.DataFrameGroupBy.apply()` only showing output from a single group when function returns an `Index` (GH28652)
- Bug in `DataFrame.groupby()` with multiple groups where an `IndexError` would be raised if any group contained all NA values (GH20519)
- Bug in `pandas.core.resample.Resampler.size()` and `pandas.core.resample.Resampler.count()` returning wrong dtype when used with an empty `Series` or `DataFrame` (GH28427)
- Bug in `DataFrame.rolling()` not allowing for rolling over datetimes when `axis=1` (GH28192)
- Bug in `DataFrame.rolling()` not allowing rolling over multi-index levels (GH15584).
- Bug in `DataFrame.rolling()` not allowing rolling on monotonic decreasing time indexes (GH19248).
- Bug in `core.groupby.DataFrameGroupBy.agg()` not able to use lambda function with named aggregation (GH27519)
- Bug in `DataFrame.groupby()` losing column name information when grouping by a categorical column (GH28787)
- Remove error raised due to duplicated input functions in named aggregation in `DataFrame.groupby()` and `Series.groupby()`. Previously error will be raised if the same function is applied on the same column and now it is allowed if new assigned names are different. (GH28426)
- `core.groupby.SeriesGroupBy.value_counts()` will be able to handle the case even when the `Grouper` makes empty groups (GH28479)
• Bug in `core.window.rolling.Rolling.quantile()` ignoring interpolation keyword argument when used within a groupby (GH28779)
• Bug in `DataFrame.groupby()` where any, all, nunique and transform functions would incorrectly handle duplicate column labels (GH21668)
• Bug in `core.groupby.DataFrameGroupBy.agg()` with timezone-aware datetime64 column incorrectly casting results to the original dtype (GH29641)
• Bug in `DataFrame.groupby()` when using axis=1 and having a single level columns index (GH30208)
• Bug in `DataFrame.groupby()` when using nunique on axis=1 (GH30253)
• Bug in `GroupBy.quantile()` with multiple list-like q value and integer column names (GH30289)
• Bug in `GroupBy.pct_change()` and `core.groupby.SeriesGroupBy.pct_change()` causes TypeError when fill_method is None (GH30463)
• Bug in `Rolling.count()` and `Expanding.count()` argument where min_periods was ignored (GH26996)

Reshaping

• Bug in `DataFrame.apply()` that caused incorrect output with empty `DataFrame` (GH28202, GH21959)
• Bug in `DataFrame.stack()` not handling non-unique indexes correctly when creating MultiIndex (GH28301)
• Bug in `pivot_table()` not returning correct type `float` when margins=True and aggfunc='mean' (GH24893)
• Fix to ensure all int dtypes can be used in `merge_asof()` when using a tolerance value. Previously every non-int64 type would raise an erroneous `MergeError` (GH28870).
• Bug in `Series.pct_change()` where supplying an anchored frequency would throw a `ValueError` (GH28664)
• Bug where `DataFrame.equals()` returned True incorrectly in some cases when two DataFrames had the same columns in different orders (GH28839)
• Bug in `Index.join()` that caused infinite recursion error for mismatched MultiIndex name orders. (GH25760, GH28956)
• Bug in `Series.pct_change()` where supplying an anchored frequency would throw a `ValueError` (GH28664)
• Better error message in `get_dummies()` when columns isn’t a list-like value (GH28383)
• Bug in `Index.join()` that caused infinite recursion error for mismatched MultiIndex name orders. (GH25760, GH28956)
• Better error message in `get_dummies()` when columns isn’t a list-like value (GH28383)
• Fix to ensure all int dtypes can be used in `merge_asof()` when using a tolerance value. Previously every non-int64 type would raise an erroneous `MergeError` (GH28870).
• Better error message in `get_dummies()` when columns isn’t a list-like value (GH28383)
• Bug in `Series.pct_change()` where supplying an anchored frequency would throw a `ValueError` (GH28664)
• Bug where `DataFrame.equals()` returned True incorrectly in some cases when two DataFrames had the same columns in different orders (GH28839)
• Bug in `DataFrame.replace()` that caused non-numeric replacer’s dtype not respected (GH26632)
• Bug in `melt()` where supplying mixed strings and numeric values for id_vars or value_vars would incorrectly raise a `ValueError` (GH29718)
• Dtypes are now preserved when transposing a `DataFrame` where each column is the same extension dtype (GH30091)
• Bug in `merge_asof()` merging on a tz-aware left_index and right_on a tz-aware column (GH29864)
• Improved error message and docstring in `cut()` and `qcut()` when labels=True (GH13318)
• Bug in missing fill_na parameter to `DataFrame.unstack()` with list of levels (GH30740)
Sparse

- Bug in SparseDataFrame arithmetic operations incorrectly casting inputs to float (GH28107)
- Bug in DataFrame.sparse returning a Series when there was a column named sparse rather than the accessor (GH30758)
- Fixed operator.xor() with a boolean-dtype SparseArray. Now returns a sparse result, rather than object dtype (GH31025)

ExtensionArray

- Bug in arrays.PandasArray when setting a scalar string (GH28118, GH28150).
- Bug where nullable integers could not be compared to strings (GH28930)
- Bug where DataFrame constructor raised ValueError with list-like data and dtype specified (GH30280)

Other

- Trying to set the display.precision, display.max_rows or display.max_columns using set_option() to anything but a None or a positive int will raise a ValueError (GH23348)
- Using DataFrame.replace() with overlapping keys in a nested dictionary will no longer raise, now matching the behavior of a flat dictionary (GH27660)
- DataFrame.to_csv() and Series.to_csv() now support dicts as compression argument with key 'method' being the compression method and others as additional compression options when the compression method is 'zip'. (GH26023)
- Bug in Series.diff() where a boolean series would incorrectly raise a TypeError (GH17294)
- Series.append() will no longer raise a TypeError when passed a tuple of Series (GH28410)
- Fix corrupted error message when calling pandas.libs._json.encode() on a 0d array (GH18878)
- Backtick quoting in DataFrame.query() and DataFrame.eval() can now also be used to use invalid identifiers like names that start with a digit, are python keywords, or are using single character operators. (GH27017)
- Bug in pd.core.util.hashing.hash_pandas_object where arrays containing tuples were incorrectly treated as non-hashable (GH28969)
- Bug in DataFrame.append() that raised IndexError when appending with empty list (GH28769)
- Fix AbstractHolidayCalendar to return correct results for years after 2030 (now goes up to 2200) (GH27790)
- Fixed IntegerArray returning inf rather than NaN for operations dividing by 0 (GH27398)
- Fixed pow operations for IntegerArray when the other value is 0 or 1 (GH29997)
- Bug in Series.count() raises if use_inf_as_na is enabled (GH29478)
- Bug in Index where a non-hashable name could be set without raising TypeError (GH29069)
- Bug in DataFrame constructor when passing a 2D ndarray and an extension dtype (GH12513)
- Bug in DataFrame.to_csv() when supplied a series with a dtype="string" and a na_rep, the na_rep was being truncated to 2 characters. (GH29975)
- Bug where DataFrame.itertuples() would incorrectly determine whether or not namedtuples could be used for dataframes of 255 columns (GH28282)
• Handle nested NumPy object arrays in `testing.assert_series_equal()` for ExtensionArray implementations (GH30841)

• Bug in `Index` constructor incorrectly allowing 2-dimensional input arrays (GH13601, GH27125)

Contributors

A total of 308 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Aaditya Panikath +
• Abdullah İhsan Seçer
• Abhijeeet Krishnan +
• Adam J. Stewart
• Adam Klaum +
• Addison Lynch
• Aivengoe +
• Alastair James +
• Albert Villanova del Moral
• Alex Kirko +
• Alfredo Granja +
• Allen Downey
• Alp Arribal +
• Andreas Buhr +
• Andrew Munch +
• Andy
• Angela Ambroz +
• Aniruddha Bhattacharjee +
• Ankit Dhankhar +
• Antonio Andraues Jr +
• Arda Kosar +
• Asish Mahapatra +
• Austin Hackett +
• Avi Kelman +
• AyowoleT +
• Bas Nijholt +
• Ben Thayer
• Bharat Raghunathan
• Bhavani Ravi
• Bhuvana KA +
• Big Head
• Blake Hawkins +
• Bobae Kim +
• Brett Naul
• Brian Wignall
• Bruno P. Kinoshita +
• Bryant Moscon +
• Cesar H +
• Chris Stadler
• Chris Zimmerman +
• Christopher Whelan
• Clemens Brunner
• Clemens Tolboom +
• Connor Charles +
• Daniel Hähnke +
• Daniel Saxton
• Darin Plutchok +
• Dave Hughes
• David Stansby
• DavidRosen +
• Dean +
• Deepan Das +
• Deepyaman Datta
• DorAmram +
• Dorothy Kabarozzi +
• Drew Heenan +
• Eliza Mae Saret +
• Elle +
• Endre Mark Borza +
• Eric Brassell +
• Eric Wong +
• Eunseop Jeong +
• Eyden Villanueva +
• Felix Divo
• ForTimeBeing +
• Francesco Truzzi +
• Gabriel Corona +
• Gabriel Monteiro +
• Galuh Sahid +
• Georgi Baychev +
• Gina
• GiuPassarelli +
• Grigorios Giannakopoulos +
• Guilherme Leite +
• Guilherme Salomé +
• Gyeongjae Choi +
• Harshavardhan Bachina +
• Harutaka Kawamura +
• Hassan Kibirige
• Hielke Walinga
• Hubert
• Hugh Kelley +
• Ian Eaves +
• Ignacio Santolin +
• Igor Filippov +
• Irv Lustig
• Isaac Virshup +
• Ivan Bessarabov +
• JMBurley +
• Jack Bicknell +
• Jacob Buckheit +
• Jan Koch
• Jan Pipek +
• Jan Škoda +
• Jan-Philip Gehrcke
• Jasper J.F. van den Bosch +
• Javad +
• Jeff Reback
• Jeremy Schendel
• Jeroen Kant +
• Jesse Pardue +
• Jethro Cao +
• Jiang Yue
• Jiaxiang +
• Jihyung Moon +
• Jimmy Callin
• Jinyang Zhou +
• Joao Victor Martinelli +
• Joaq Almirante +
• John G Evans +
• John Ward +
• Jonathan Larkin +
• Joris Van den Bossche
• Josh Dimarsky +
• Joshua Smith +
• Josiah Baker +
• Julia Signell +
• Jung Dong Ho +
• Justin Cole +
• Justin Zheng
• Kaiqi Dong
• Karthigeyan +
• Katherine Younglove +
• Katrin Leinweber
• Kee Chong Tan +
• Keith Kraus +
• Kevin Nguyen +
• Kevin Sheppard
• Kisekka David +
• Koushik +
• Kyle Boone +
• Kyle Mc Cahill +
• Laura Collard, PhD +
• LiuSeeker +
• Louis Huynh +
• Lucas Scarlato Astur +
• Luiz Gustavo +
• Luke +
• Luke Shepard +
• MKhalusova +
• Mabel Villalba
• Maciej J +
• Mak Sze Chun
• Manu NALEPA +
• Marc
• Marc Garcia
• Marco Gorelli +
• Marco Neumann +
• Martin Winkel +
• Martina G. Vilas +
• Mateusz +
• Matthew Roeschke
• Matthew Tan +
• Max Bolingbroke
• Max Chen +
• MeeseeksMachine
• Miguel +
• MinGyo Jung +
• Mohamed Amine ZGHAL +
• Mohit Anand +
• MomIsBestFriend +
• Naomi Bonnin +
• Nathan Abel +
• Nico Cernek +
• Nigel Markey +
• Noritada Kobayashi +
• Oktay Sabak +
• Oliver Hofkens +
• Oluokun Adedayo +
• Osman +
• Oğuzhan Öğreden +
• Pandas Development Team +
• Patrik Hlobil +
• Paul Lee +
• Paul Siegel +
• Petr Baev +
• Pietro Battiston
• Prakhar Pandey +
• Puneeth K +
• Raghav +
• Rajat +
• Rajhans Jadhao +
• Rajiv Bharadwaj +
• Rik-de-Kort +
• Roei.r
• Rohit Sanjay +
• Ronan Lamy +
• Roshni +
• Roymprog +
• Rushabh Vasani +
• Ryan Grout +
• Ryan Nazareth
• Samesh Lakhotia +
• Samuel Sinayoko
• Samyak Jain +
• Sarah Donehower +
• Sarah Masud +
• Saul Shanabrook +
• Scott Cole +
• SdgJlbl +
• Seb +
• Sergei Ivko +
• Shadi Akiki
• Shorokhov Sergey
• Siddhesh Poyarekar +
• Sidharthan Nair +
• Simon Gibbons
• Simon Hawkins
• Simon-Martin Schröder +
• Sofiane Mahiou +
• Sourav kumar +
• Souvik Mandal +
• Soyoun Kim +
• Sparkle Russell-Puleri +
• Srinivas Reddy Thatiparthi ( )
• Stuart Berg +
• Sumanau Sareen
• Szymon Bednarek +
• Tambe Tabitha Achere +
• Tan Tran
• Tang Heyi +
• Tanmay Daripa +
• Tanya Jain
• Terji Petersen
• Thomas Li +
• Tirth Jain +
• Tola A +
• Tom Augspurger
• Tommy Lynch +
• Tomoyuki Suzuki +
• Tony Lorenzo
• Unprocessable +
• Uwe L. Korn
• Vaibhav Vishal
• Victoria Zdanovskaya +
• Vijayant +
• Vishwak Srinivasan +
• WANG Aiyong
• Wenhuan
• Wes McKinney
• Will Ayd
• Will Holmgren
• William Ayd
• William Blan +
• Wouter Overmeire
• Wuraola Oyewusi +
<table>
<thead>
<tr>
<th>User Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>YaOz1</td>
</tr>
<tr>
<td>Yash Shukla</td>
</tr>
<tr>
<td>Yu Wang</td>
</tr>
<tr>
<td>Yusei Tahara</td>
</tr>
<tr>
<td>alexander135</td>
</tr>
<tr>
<td>alimcmaster1</td>
</tr>
<tr>
<td>avelineg</td>
</tr>
<tr>
<td>bganglia</td>
</tr>
<tr>
<td>bolkedebriu</td>
</tr>
<tr>
<td>bravech</td>
</tr>
<tr>
<td>chinhwee</td>
</tr>
<tr>
<td>cruzzoe</td>
</tr>
<tr>
<td>dalgarno</td>
</tr>
<tr>
<td>daniellebrown</td>
</tr>
<tr>
<td>danielplawrence</td>
</tr>
<tr>
<td>est271</td>
</tr>
<tr>
<td>francisco souza</td>
</tr>
<tr>
<td>ganevgv</td>
</tr>
<tr>
<td>garanews</td>
</tr>
<tr>
<td>gfyoung</td>
</tr>
<tr>
<td>h-vetinari</td>
</tr>
<tr>
<td>hasnain2808</td>
</tr>
<tr>
<td>ianzur</td>
</tr>
<tr>
<td>jalbritt</td>
</tr>
<tr>
<td>jbrockmendel</td>
</tr>
<tr>
<td>jeschwar</td>
</tr>
<tr>
<td>jlamborn324</td>
</tr>
<tr>
<td>joy-rosie</td>
</tr>
<tr>
<td>kernc</td>
</tr>
<tr>
<td>killerontherun1</td>
</tr>
<tr>
<td>krey</td>
</tr>
<tr>
<td>lexy-lixinyu</td>
</tr>
<tr>
<td>lucyleeow</td>
</tr>
<tr>
<td>lukasbk</td>
</tr>
<tr>
<td>maheshbapatu</td>
</tr>
<tr>
<td>mck619</td>
</tr>
</tbody>
</table>
5.6 Version 0.25

5.6.1 What’s new in 0.25.3 (October 31, 2019)

These are the changes in pandas 0.25.3. See Release notes for a full changelog including other versions of pandas.
Bug fixes

GroupBy/resample/rolling

- Bug in `DataFrameGroupBy.quantile()` where NA values in the grouping could cause segfaults or incorrect results (GH28882)

Contributors

A total of 2 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Will Ayd
- William Ayd

5.6.2 What’s new in 0.25.2 (October 15, 2019)

These are the changes in pandas 0.25.2. See Release notes for a full changelog including other versions of pandas.

Note: pandas 0.25.2 adds compatibility for Python 3.8 (GH28147).

Bug fixes

Indexing

- Fix regression in `DataFrame.reindex()` not following the limit argument (GH28631).
- Fix regression in `RangeIndex.get_indexer()` for decreasing `RangeIndex` where target values may be improperly identified as missing/present (GH28678)

IO

- Fix regression in notebook display where `<th>` tags were missing for `DataFrame.index` values (GH28204).
- Regression in `to_csv()` where writing a `Series` or `DataFrame` indexed by an `IntervalIndex` would incorrectly raise a `TypeError` (GH28210)
- Fix `to_csv()` with `ExtensionArray` with list-like values (GH28840).

GroupBy/resample/rolling

- Bug incorrectly raising an `IndexError` when passing a list of quantiles to `pandas.core.groupby.DataFrameGroupBy.quantile()` (GH28113).
- Bug in `pandas.core.groupby.GroupBy.shift()`, `pandas.core.groupby.GroupBy.bfill()` and `pandas.core.groupby.GroupBy.ffill()` where timezone information would be dropped (GH19995, GH27992)
Other

• Compatibility with Python 3.8 in DataFrame.query() (GH27261)
• Fix to ensure that tab-completion in an IPython console does not raise warnings for deprecated attributes (GH27900).

Contributors

A total of 6 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Felix Divo +
• Jeremy Schendel
• Joris Van den Bossche
• MeeseeksMachine
• Tom Augspurger
• jbrockmendel

5.6.3 What’s new in 0.25.1 (August 21, 2019)

These are the changes in pandas 0.25.1. See Release notes for a full changelog including other versions of pandas.

IO and LZMA

Some users may unknowingly have an incomplete Python installation lacking the lzma module from the standard library. In this case, import pandas failed due to an ImportError (GH27575). pandas will now warn, rather than raising an ImportError if the lzma module is not present. Any subsequent attempt to use lzma methods will raise a RuntimeError. A possible fix for the lack of the lzma module is to ensure you have the necessary libraries and then re-install Python. For example, on MacOS installing Python with pyenv may lead to an incomplete Python installation due to unmet system dependencies at compilation time (like xz). Compilation will succeed, but Python might fail at run time. The issue can be solved by installing the necessary dependencies and then re-installing Python.

Bug fixes

Categorical

• Bug in Categorical.fillna() that would replace all values, not just those that are NaN (GH26215)
pandas: powerful Python data analysis toolkit, Release 1.4.2

Datetimelike

- Bug in `to_datetime()` where passing a timezone-naive `DatetimeArray` or `DatetimeIndex` and `utc=True` would incorrectly return a timezone-naive result (GH27733)
- Bug in `Period.to_timestamp()` where a `Period` outside the `Timestamp` implementation bounds (roughly 1677-09-21 to 2262-04-11) would return an incorrect `Timestamp` instead of raising `OutOfBoundsDatetime` (GH19643)
- Bug in iterating over `DatetimeIndex` when the underlying data is read-only (GH28055)

Timezones

- Bug in `Index` where a numpy object array with a timezone aware `Timestamp` and `np.nan` would not return a `DatetimeIndex` (GH27011)

Numeric

- Bug in `Series.interpolate()` when using a timezone aware `DatetimeIndex` (GH27548)
- Bug when printing negative floating point complex numbers would raise an `IndexError` (GH27484)
- Bug where `DataFrame` arithmetic operators such as `DataFrame.mul()` with a `Series` with `axis=1` would raise an `AttributeError` on `DataFrame` larger than the minimum threshold to invoke `numexpr` (GH27636)
- Bug in `DataFrame` arithmetic where missing values in results were incorrectly masked with `NaN` instead of `Inf` (GH27464)

Conversion

- Improved the warnings for the deprecated methods `Series.real()` and `Series.imag()` (GH27610)

Interval

- Bug in `IntervalIndex` where `dir(obj)` would raise `ValueError` (GH27571)

Indexing

- Bug in partial-string indexing returning a NumPy array rather than a `Series` when indexing with a scalar like `.loc['2015']` (GH27516)
- Break reference cycle involving `Index` and other index classes to allow garbage collection of index objects without running the GC. (GH27585, GH27840)
- Fix regression in assigning values to a single column of a `DataFrame` with a `MultiIndex` columns (GH27841).
- Fix regression in `.ix` fallback with an `IntervalIndex` (GH27865).
Missing

- Bug in `pandas.isnull()` or `pandas.isna()` when the input is a type e.g. `type(pandas.Series())` (GH27482)

IO

- Avoid calling `S3File.s3` when reading parquet, as this was removed in s3fs version 0.3.0 (GH27756)
- Better error message when a negative header is passed in `pandas.read_csv()` (GH27779)
- Follow the `min_rows` display option (introduced in v0.25.0) correctly in the HTML repr in the notebook (GH27991).

Plotting

- Added a `pandas_plotting_backends` entrypoint group for registering plot backends. See Plotting backends for more (GH26747).
- Fixed the re-instatement of Matplotlib datetime converters after calling `pandas.plotting.deregister_matplotlib_converters()` (GH27481).
- Fix compatibility issue with matplotlib when passing a pandas Index to a plot call (GH27775).

GroupBy/resample/rolling

- Fixed regression in `pandas.core.groupby.DataFrameGroupBy.quantile()` raising when multiple quantiles are given (GH27526)
- Bug in `pandas.core.groupby.DataFrameGroupBy.transform()` where applying a timezone conversion lambda function would drop timezone information (GH27496)
- Bug in `pandas.core.groupby.GroupBy.nth()` where observed=False was being ignored for Categorical groupers (GH26385)
- Bug in windowing over read-only arrays (GH27766)
- Fixed segfault in `pandas.core.groupby.DataFrameGroupBy.quantile` when an invalid quantile was passed (GH27470)

Reshaping

- A KeyError is now raised if `.unstack()` is called on a Series or DataFrame with a flat Index passing a name which is not the correct one (GH18303)
- Bug `merge_asof()` could not merge Timedelta objects when passing tolerance kwarg (GH27642)
- Bug in `DataFrame.crosstab()` when margins set to True and normalize is not False, an error is raised. (GH27500)
- `DataFrame.join()` now suppresses the FutureWarning when the sort parameter is specified (GH21952)
- Bug in `DataFrame.join()` raising with readonly arrays (GH27943)
Sparse

- Bug in reductions for Series with Sparse dtypes (GH27080)

Other

- Bug in Series.replace() and DataFrame.replace() when replacing timezone-aware timestamps using a dict-like replacer (GH27720)
- Bug in Series.rename() when using a custom type indexer. Now any value that isn't callable or dict-like is treated as a scalar. (GH27814)

Contributors

A total of 5 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Jeff Reback
- Joris Van den Bossche
- MeeseeksMachine +
- Tom Augspurger
- jbrockmendel

5.6.4 What's new in 0.25.0 (July 18, 2019)

**Warning:** Starting with the 0.25.x series of releases, pandas only supports Python 3.5.3 and higher. See Dropping Python 2.7 for more details.

**Warning:** The minimum supported Python version will be bumped to 3.6 in a future release.

**Warning:** Panel has been fully removed. For N-D labeled data structures, please use xarray

**Warning:** read_pickle() and read_msgpack() are only guaranteed backwards compatible back to pandas version 0.20.3 (GH27082)

These are the changes in pandas 0.25.0. See Release notes for a full changelog including other versions of pandas.
Enhancements

GroupBy aggregation with relabeling

pandas has added special groupby behavior, known as “named aggregation”, for naming the output columns when applying multiple aggregation functions to specific columns (GH18366, GH26512).

```
In [1]: animals = pd.DataFrame({
    ...:     'kind': ['cat', 'dog', 'cat', 'dog'],
    ...:     'height': [9.1, 6.0, 9.5, 34.0],
    ...:     'weight': [7.9, 7.5, 9.9, 198.0]})

In [2]: animals
Out[2]:
    kind height  weight
 0    cat     9.1    7.9
 1    dog     6.0    7.5
 2    cat     9.5    9.9
 3    dog    34.0   198.0
[4 rows x 3 columns]
```

```
In [3]: animals.groupby("kind").agg(
    ...:     min_height=pd.NamedAgg(column='height', aggfunc='min'),
    ...:     max_height=pd.NamedAgg(column='height', aggfunc='max'),
    ...:     average_weight=pd.NamedAgg(column='weight', aggfunc=np.mean),
    ...: )

Out[3]:
    min_height  max_height  average_weight
  kind
  cat  9.1      9.5          8.90
  dog  6.0     34.0         102.75
[2 rows x 3 columns]
```

Pass the desired columns names as the **kwargs to `.agg`. The values of **kwargs should be tuples where the first element is the column selection, and the second element is the aggregation function to apply. `pandas` provides the `pandas.NamedAgg` namedtuple to make it clearer what the arguments to the function are, but plain tuples are accepted as well.

```
In [4]: animals.groupby("kind").agg(
    ...:     min_height=('height', 'min'),
    ...:     max_height=('height', 'max'),
    ...:     average_weight=('weight', np.mean),
    ...: )

Out[4]:
    min_height  max_height  average_weight
  kind
  cat  9.1      9.5          8.90
  dog  6.0     34.0         102.75
```

(continues on next page)
Named aggregation is the recommended replacement for the deprecated “dict-of-dicts” approach to naming the output of column-specific aggregations (Deprecate groupby.agg() with a dictionary when renaming).

A similar approach is now available for Series groupby objects as well. Because there’s no need for column selection, the values can just be the functions to apply.

```python
In [5]: animals.groupby("kind").height.agg(
             ...:     min_height="min",
             ...:     max_height="max",
             ...:     )
...:
Out[5]:
    min_height  max_height
kind     
    cat  9.1      9.5
    dog  6.0     34.0

[2 rows x 2 columns]
```

This type of aggregation is the recommended alternative to the deprecated behavior when passing a dict to a Series groupby aggregation (Deprecate groupby.agg() with a dictionary when renaming).

See Named aggregation for more.

### GroupBy aggregation with multiple lambdas

You can now provide multiple lambda functions to a list-like aggregation in pandas.core.groupby.GroupBy.agg (GH26430).

```python
In [6]: animals.groupby('kind').height.agg([
             ...:     lambda x: x.iloc[0],
             ...:     lambda x: x.iloc[-1]
             ...: ])
...:
Out[6]:
    <lambda_0>  <lambda_1>
kind     
    cat   9.1     9.5
    dog   6.0    34.0

[2 rows x 2 columns]
```

```python
In [7]: animals.groupby('kind').agg([
             ...:     lambda x: x.iloc[0] - x.iloc[1],
             ...:     lambda x: x.iloc[0] + x.iloc[1]
             ...: ])
...:
Out[7]:
    height  weight
    <lambda_0> <lambda_1> <lambda_0> <lambda_1>
kind     
    (continues on next page)
Previously, these raised a `SpecificationError`.

**Better repr for MultiIndex**

Printing of `MultiIndex` instances now shows tuples of each row and ensures that the tuple items are vertically aligned, so it’s now easier to understand the structure of the `MultiIndex` ([GH13480](https://github.com/pandas-dev/pandas/issues/13480)).

The repr now looks like this:

```python
In [8]: pd.MultiIndex.from_product([['a', 'abc'], range(500)])
Out[8]:
MultiIndex([('a', 0),
            ('a', 1),
            ('a', 2),
            ('a', 3),
            ('a', 4),
            ('a', 5),
            ('a', 6),
            ('a', 7),
            ('a', 8),
            ('a', 9),
            ...
            ('abc', 490),
            ('abc', 491),
            ('abc', 492),
            ('abc', 493),
            ('abc', 494),
            ('abc', 495),
            ('abc', 496),
            ('abc', 497),
            ('abc', 498),
            ('abc', 499)],
            length=1000)
```

Previously, outputting a `MultiIndex` printed all the levels and codes of the `MultiIndex`, which was visually unappealing and made the output more difficult to navigate. For example (limiting the range to 5):

```python
In [1]: pd.MultiIndex.from_product([['a', 'abc'], range(5)])
Out[1]: MultiIndex(levels=[['a', 'abc'], [0, 1, 2, 3]],
            ...
codes=[[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])
```

In the new repr, all values will be shown, if the number of rows is smaller than `options.display.max_seq_items` (default: 100 items). Horizontally, the output will truncate, if it’s wider than `options.display.width` (default: 80 characters).
**Shorter truncated repr for Series and DataFrame**

Currently, the default display options of pandas ensure that when a Series or DataFrame has more than 60 rows, its repr gets truncated to this maximum of 60 rows (the `display.max_rows` option). However, this still gives a repr that takes up a large part of the vertical screen estate. Therefore, a new option `display.min_rows` is introduced with a default of 10 which determines the number of rows showed in the truncated repr:

- For small Series or DataFrames, up to `max_rows` number of rows is shown (default: 60).
- For larger Series of DataFrame with a length above `max_rows`, only `min_rows` number of rows is shown (default: 10, i.e. the first and last 5 rows).

This dual option allows to still see the full content of relatively small objects (e.g. `df.head(20)` shows all 20 rows), while giving a brief repr for large objects.

To restore the previous behaviour of a single threshold, set `pd.options.display.min_rows = None`.

**JSON normalize with max_level param support**

`json_normalize()` normalizes the provided input dict to all nested levels. The new `max_level` parameter provides more control over which level to end normalization (GH23843):

The repr now looks like this:

```python
from pandas.io.json import json_normalize

data = [{
    'CreatedBy': {'Name': 'User001'},
    'Lookup': {'TextField': 'Some text',
               'UserField': {'Id': 'ID001', 'Name': 'Name001'}},
    'Image': {'a': 'b'}
}]
json_normalize(data, max_level=1)
```

**Series.explode to split list-like values to rows**

`Series` and `DataFrame` have gained the `DataFrame.explode()` methods to transform list-likes to individual rows. See section on Exploding list-like column in docs for more information (GH16538, GH10511)

Here is a typical usecase. You have comma separated string in a column.

```python
In [9]: df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1},
                        {'var1': 'd,e,f', 'var2': 2}])
```

Creating a long form `DataFrame` is now straightforward using chained operations
Other enhancements

- `DataFrame.plot()` keywords logy, logx and loglog can now accept the value 'sym' for symlog scaling. (GH24867)
- Added support for ISO week year format ('%G-%V-%u') when parsing datetimes using `to_datetime()` (GH16607)
- Indexing of `DataFrame` and `Series` now accepts zerodim `np.ndarray` (GH24919)
- `Timestamp.replace()` now supports the `fold` argument to disambiguate DST transition times (GH25017)
- `DataFrame.at_time()` and `Series.at_time()` now support `datetime.time` objects with timezones (GH24043)
- `DataFrame.pivot_table()` now accepts an `observed` parameter which is passed to underlying calls to `DataFrame.groupby()` to speed up grouping categorical data. (GH24923)
- `Series.str` has gained `Series.str.casefold()` method to removes all case distinctions present in a string (GH25405)
- `DataFrame.set_index()` now works for instances of `abc.Iterator`, provided their output is of the same length as the calling frame (GH22484, GH24984)
- `DatetimeIndex.union()` now supports the `sort` argument. The behavior of the sort parameter matches that of `Index.union()` (GH24994)
- `RangeIndex.union()` now supports the sort argument. If sort=False an unsorted `Int64Index` is always returned. sort=None is the default and returns a monotonically increasing `RangeIndex` if possible or a sorted `Int64Index` if not (GH24471)
- `TimedeltaIndex.intersection()` now also supports the sort keyword (GH24471)
- `DataFrame.rename()` now supports the `errors` argument to raise errors when attempting to rename nonexistent keys (GH13473)
- Added `Sparse accessor` for working with a `DataFrame` whose values are sparse (GH25681)
- `RangeIndex` has gained `start`, `stop`, and `step` attributes (GH25710)
- `datetime.timezone` objects are now supported as arguments to timezone methods and constructors (GH25065)
- `DataFrame.query()` and `DataFrame.eval()` now supports quoting column names with backticks to refer to names with spaces (GH6508)
- `merge_asof()` now gives a more clear error message when merge keys are categoricals that are not equal (GH26136)
• pandas.core.window.Rolling() supports exponential (or Poisson) window type (GH1303)
• Error message for missing required imports now includes the original import error’s text (GH23868)
• DatetimeIndex and TimedeltaIndex now have a mean method (GH24757)
• DataFrame.describe() now formats integer percentiles without decimal point (GH26660)
• Added support for reading SPSS .sav files using read_spss() (GH26537)
• Added new option plotting.backend to be able to select a plotting backend different than the existing matplotlib one. Use pandas.set_option('plotting.backend', '<backend-module>') where <backend-module> is a library implementing the pandas plotting API (GH14130)
• pandas.offsets.BusinessHour supports multiple opening hours intervals (GH15481)
• read_excel() can now use openpyxl to read Excel files via the engine='openpyxl' argument. This will become the default in a future release (GH11499)
• pandas.io.excel.read_excel() supports reading OpenDocument tables. Specify engine='odf' to enable. Consult the IO User Guide for more details (GH9070)
• Interval, IntervalIndex, and IntervalArray have gained an is_empty attribute denoting if the given interval(s) are empty (GH27219)

Backwards incompatible API changes

Indexing with date strings with UTC offsets

Indexing a DataFrame or Series with a DatetimeIndex with a date string with a UTC offset would previously ignore the UTC offset. Now, the UTC offset is respected in indexing. (GH24076, GH16785)

```python
In [12]: df = pd.DataFrame([0], index=pd.DatetimeIndex(['2019-01-01', tz='US/Pacific']))

In [13]: df
Out[13]:
   2019-01-01 00:00:00-08:00  0
[1 rows x 1 columns]

Previous behavior:

```python
In [3]: df['2019-01-01 00:00:00+04:00':'2019-01-01 01:00:00+04:00']
Out[3]:
   2019-01-01 00:00:00-08:00  0

New behavior:

```python
In [14]: df['2019-01-01 12:00:00+04:00':'2019-01-01 13:00:00+04:00']
Out[14]:
   2019-01-01 00:00:00-08:00  0
[1 rows x 1 columns]

5.6. Version 0.25
MultiIndex constructed from levels and codes

Constructing a MultiIndex with NaN levels or codes value < -1 was allowed previously. Now, construction with codes value < -1 is not allowed and NaN levels’ corresponding codes would be reassigned as -1. (GH19387)

Previous behavior:

```python
In [1]: pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
                           codes=[[0, -1, 1, 2, 3, 4]])
Out[1]: MultiIndex(levels=[[nan, None, NaT, 128, 2]],
                      codes=[[0, -1, 1, 2, 3, 4]])
In [2]: pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
Out[2]: MultiIndex(levels=[[1, 2]],
                      codes=[[0, -2]])
```

New behavior:

```python
In [15]: pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
                           codes=[[0, -1, 1, 2, 3, 4]])
Out[15]: MultiIndex((nan,)
                    (nan,)
                    (nan,)
                    (nan,)
                    (128,)
                    (2,))
In [16]: pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [16], in <cell line: 1>()
----> 1 pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
```

File /pandas/pandas/core/indexes/multi.py:343, in MultiIndex.__new__(cls, levels, codes, ...
   ...sortorder, names, dtype, copy, name, verify_integrity)
   340 result.sortorder = sortorder
   342 if verify_integrity:
--> 343    new_codes = result._verify_integrity()
   344 result._codes = new_codes
   346 result._reset_identity()

File /pandas/pandas/core/indexes/multi.py:416, in MultiIndex._verify_integrity(self, ...
   ...codes, levels)
   410     raise ValueError(f"On level \{i\}, code value \{level_codes.min\() \< -1\"
   411     f"On level \{i\}, code max \{level_codes.max\() \>= length of ",
   412     f"level \{i\}]. NOTE: this index is in an "
   413     "inconsistent state"
   414 )
   415 if len(level_codes) and level_codes.min() < -1:
```

(continues on next page)
417 if not level.is_unique:
418     raise ValueError(
419         f"Level values must be unique: {list(level)} on level {i}"
420     )

ValueError: On level 0, code value (-2) < -1

**GroupBy.apply on DataFrame evaluates first group only once**

The implementation of DataFrameGroupBy.apply() previously evaluated the supplied function consistently twice on the first group to infer if it is safe to use a fast code path. Particularly for functions with side effects, this was an undesired behavior and may have led to surprises. (GH2936, GH2656, GH7739, GH10519, GH12155, GH20084, GH21417)

Now every group is evaluated only a single time.

```python
In [17]: df = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})
In [18]: df
Out[18]:
   a b
0  x 1
1  y 2
[2 rows x 2 columns]
In [19]: def func(group):
       ...
       ....:     print(group.name)
       ....:     return group
       ....:

Previous behavior:

In [3]: df.groupby('a').apply(func)
   x
   x
   y
Out[3]:
   a b
0  x 1
1  y 2

New behavior:

In [20]: df.groupby("a").apply(func)
   x
   y
Out[20]:
   a b
0  x 1
1  y 2
```

(continues on next page)
Concatenating sparse values

When passed DataFrames whose values are sparse, `concat()` will now return a `Series` or `DataFrame` with sparse values, rather than a `SparseDataFrame` (GH25702).

```python
In [21]: df = pd.DataFrame({'A': pd.SparseArray([0, 1])})
```

**Previous behavior:**

```python
In [2]: type(pd.concat([df, df]))
pandas.core.sparse.frame.SparseDataFrame
```

**New behavior:**

```python
In [22]: type(pd.concat([df, df]))
Out[22]: pandas.core.frame.DataFrame
```

This now matches the existing behavior of `concat` on `Series` with sparse values. `concat()` will continue to return a `SparseDataFrame` when all the values are instances of `SparseDataFrame`.

This change also affects routines using `concat()` internally, like `get_dummies()`, which now returns a `DataFrame` in all cases (previously a `SparseDataFrame` was returned if all the columns were dummy encoded, and a `DataFrame` otherwise).

Providing any `SparseSeries` or `SparseDataFrame` to `concat()` will cause a `SparseSeries` or `SparseDataFrame` to be returned, as before.

The `.str`-accessor performs stricter type checks

Due to the lack of more fine-grained dtypes, `Series.str` so far only checked whether the data was of `object` dtype. `Series.str` will now infer the dtype data within the Series; in particular, 'bytes'-only data will raise an exception (except for `Series.str.decode()`, `Series.str.get()`, `Series.str.len()`, `Series.str.slice()`), see GH23163, GH23011, GH23551.

**Previous behavior:**

```python
In [1]: s = pd.Series(np.array(['a', 'ba', 'cba'], 'S'), dtype=object)
```

```python
In [2]: s
Out[2]:
    0   b'a'
    1   b'ba'
    2   b'cba'
 dtype: object
```

```python
In [3]: s.str.startswith(b'a')
Out[3]:
    0   True
    1  False
```

(continues on next page)
New behavior:

```python
In [23]: s = pd.Series(np.array(['a', 'ba', 'cba'], 'S'), dtype=object)
```

```python
In [24]: s
Out[24]:
0  'a'
1  'ba'
2  'cba'
Length: 3, dtype: object
```

```python
In [25]: s.str.startswith('a')
---------------------------------------------------------------------------
TypeError
Input In [25], in <cell line: 1>()
----> 1 s.str.startswith(b'a')

File /pandas/pandas/core/strings/accessor.py:124, in forbid_nonstring_types.<locals>._forbid_nonstring_types.<locals>.wrapper(self, *args, **kwargs)
     119     if self._inferred_dtype not in allowed_types:
     120         msg = (
     121             f"Cannot use .str.{func_name} with values of "
     122             f"{self._inferred_dtype}"."
     123     )
---> 124     raise TypeError(msg)
     125     return func(self, *args, **kwargs)
```

```output
TypeError: Cannot use .str.startswith with values of inferred dtype 'bytes'.
```

Categorical dtypes are preserved during GroupBy

Previously, columns that were categorical, but not the groupby key(s) would be converted to object dtype during groupby operations. pandas now will preserve these dtypes. (GH18502)

```python
In [26]: cat = pd.Categorical(['foo', 'bar', 'bar', 'qux'], ordered=True)
In [27]: df = pd.DataFrame({'payload': [-1, -2, -1, -2], 'col': cat})
In [28]: df
Out[28]:
            payload   col
0          -1    foo
1           2    bar
2           2    bar
3           3    qux
[4 rows x 2 columns]
```

(continues on next page)
In [29]: df.dtypes
Out[29]:
payload    int64
col      category
Length: 2, dtype: object

Previous Behavior:

In [5]: df.groupby('payload').first().col.dtype
Out[5]: dtype('O')

New Behavior:

In [30]: df.groupby('payload').first().col.dtype
Out[30]: CategoricalDtype(categories=['bar', 'foo', 'qux'], ordered=True)

Incompatible Index type unions

When performing `Index.union()` operations between objects of incompatible dtypes, the result will be a base `Index` of dtype object. This behavior holds true for unions between `Index` objects that previously would have been prohibited. The dtype of empty `Index` objects will now be evaluated before performing union operations rather than simply returning the other `Index` object. `Index.union()` can now be considered commutative, such that \(A.union(B) = B.union(A)\) (GH23525).

Previous behavior:

In [1]: pd.period_range('19910905', periods=2).union(pd.Int64Index([1, 2, 3]))
...  
ValueError: can only call with other PeriodIndex-ed objects

In [2]: pd.Index([], dtype=object).union(pd.Index([1, 2, 3]))
Out[2]: Int64Index([1, 2, 3], dtype='int64')

New behavior:

In [3]: pd.period_range('19910905', periods=2).union(pd.Int64Index([1, 2, 3]))
Out[3]: Index([1991-09-05, 1991-09-06, 1, 2, 3], dtype='object')

In [4]: pd.Index([], dtype=object).union(pd.Index([1, 2, 3]))
Out[4]: Index([1, 2, 3], dtype='object')

Note that integer- and floating-dtype indexes are considered “compatible”. The integer values are coerced to floating point, which may result in loss of precision. See Set operations on Index objects for more.
DataFrame GroupBy ffill/bfill no longer return group labels

The methods ffill, bfill, pad and backfill of DataFrameGroupBy previously included the group labels in the return value, which was inconsistent with other groupby transforms. Now only the filled values are returned. (GH21521)

```
In [31]: df = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})
In [32]: df
Out[32]:
   a b
0  x 1
1  y 2
[2 rows x 2 columns]
```

*Previous behavior:*

```
In [3]: df.groupby("a").ffill()
Out[3]:
   a b
0  x 1
1  y 2
```

*New behavior:*

```
In [33]: df.groupby("a").ffill()
Out[33]:
   b
0 1
1 2
[2 rows x 1 columns]
```

DataFrame describe on an empty Categorical / object column will return top and freq

When calling DataFrame.describe() with an empty categorical / object column, the ‘top’ and ‘freq’ columns were previously omitted, which was inconsistent with the output for non-empty columns. Now the ‘top’ and ‘freq’ columns will always be included, with numpy.nan in the case of an empty DataFrame (GH26397)

```
In [34]: df = pd.DataFrame({"empty_col": pd.Categorical([])})
In [35]: df
Out[35]:
Empty DataFrame
Columns: [empty_col]
Index: []
[0 rows x 1 columns]
```

*Previous behavior:*

```
```
In [3]: df.describe()
Out[3]:
   empty_col
count   0
unique   0

New behavior:

In [36]: df.describe()
Out[36]:
   empty_col
count   0
unique   0
top    NaN
freq   NaN
[4 rows x 1 columns]

__str__ methods now call __repr__ rather than vice versa

pandas has until now mostly defined string representations in a pandas objects’ __str__/__unicode__/__bytes__ methods, and called __str__ from the __repr__ method, if a specific __repr__ method is not found. This is not needed for Python3. In pandas 0.25, the string representations of pandas objects are now generally defined in __repr__, and calls to __str__ in general now pass the call on to the __repr__, if a specific __str__ method doesn’t exist, as is standard for Python. This change is backward compatible for direct usage of pandas, but if you subclass pandas objects and give your subclasses specific __str__/__repr__ methods, you may have to adjust your __str__/__repr__ methods (GH26495).

Indexing an IntervalIndex with Interval objects

Indexing methods for IntervalIndex have been modified to require exact matches only for Interval queries. IntervalIndex methods previously matched on any overlapping Interval. Behavior with scalar points, e.g. querying with an integer, is unchanged (GH16316).

In [37]: ii = pd.IntervalIndex.from_tuples([(0, 4), (1, 5), (5, 8)])

In [38]: ii
Out[38]: IntervalIndex([(0, 4), (1, 5), (5, 8)], dtype='interval[int64, right]')

The in operator (__contains__) now only returns True for exact matches to Intervals in the IntervalIndex, whereas this would previously return True for any Interval overlapping an Interval in the IntervalIndex.

Previous behavior:

In [4]: pd.Interval(1, 2, closed='neither') in ii
Out[4]: True

In [5]: pd.Interval(-10, 10, closed='both') in ii
Out[5]: True

New behavior:
The `get_loc()` method now only returns locations for exact matches to `Interval` queries, as opposed to the previous behavior of returning locations for overlapping matches. A `KeyError` will be raised if an exact match is not found.

**Previous behavior:**

```python
In [6]: ii.get_loc(pd.Interval(1, 5))
Out[6]: array([0, 1])

In [7]: ii.get_loc(pd.Interval(2, 6))
Out[7]: array([0, 1, 2])
```

**New behavior:**

```python
In [6]: ii.get_loc(pd.Interval(1, 5))
Out[6]: 1

In [7]: ii.get_loc(pd.Interval(2, 6))
---------------------------------------------------------------------------
KeyError: Interval(2, 6, closed='right')
```

Likewise, `get_indexer()` and `get_indexer_non_unique()` will also only return locations for exact matches to `Interval` queries, with -1 denoting that an exact match was not found.

These indexing changes extend to querying a `Series` or `DataFrame` with an `IntervalIndex` index.

```python
In [41]: s = pd.Series(list('abc'), index=ii)

In [42]: s
Out[42]:
(0, 4]   a
(1, 5]   b
(5, 8]   c
Length: 3, dtype: object
```

Selecting from a `Series` or `DataFrame` using `[]`(`__getitem__`) or `loc` now only returns exact matches for `Interval` queries.

**Previous behavior:**

```python
In [8]: s[pd.Interval(1, 5)]
Out[8]:
(0, 4]   a
(1, 5]   b
dtype: object

In [9]: s.loc[pd.Interval(1, 5)]
Out[9]:
(0, 4]   a
(continues on next page)```
(continued from previous page)

(1, 5]  b
dtype: object

New behavior:

In [43]: s[pd.Interval(1, 5)]
Out[43]: 'b'

In [44]: s.loc[pd.Interval(1, 5)]
Out[44]: 'b'

Similarly, a KeyError will be raised for non-exact matches instead of returning overlapping matches.

Previous behavior:

In [9]: s[pd.Interval(2, 3)]
Out[9]:
(0, 4]  a
(1, 5]  b
dtype: object

In [10]: s.loc[pd.Interval(2, 3)]
Out[10]:
(0, 4]  a
(1, 5]  b
dtype: object

New behavior:

In [6]: s[pd.Interval(2, 3)]

KeyError: Interval(2, 3, closed='right')

In [7]: s.loc[pd.Interval(2, 3)]

KeyError: Interval(2, 3, closed='right')

The overlaps() method can be used to create a boolean indexer that replicates the previous behavior of returning overlapping matches.

New behavior:

In [45]: idxr = s.index.overlaps(pd.Interval(2, 3))

In [46]: idxr
Out[46]: array([ True,  True, False])

In [47]: s[idxr]
Out[47]:
(0, 4]  a
(1, 5]  b
Length: 2, dtype: object

In [48]: s.loc[idxr]

(continues on next page)
Binary ufuncs on Series now align

Applying a binary ufunc like `numpy.power()` now aligns the inputs when both are `Series` (GH23293).

```
In [49]: s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
In [50]: s2 = pd.Series([3, 4, 5], index=['d', 'c', 'b'])
In [51]: s1
Out[51]:
   a 1
   b 2
   c 3
Length: 3, dtype: int64
In [52]: s2
Out[52]:
   d 3
   c 4
   b 5
Length: 3, dtype: int64

Previous behavior

In [5]: np.power(s1, s2)
Out[5]:
   a  1
   b 16
   c 243
dtype: int64

New behavior

In [53]: np.power(s1, s2)
Out[53]:
   a  1.0
   b 32.0
   c 81.0
d  NaN
Length: 4, dtype: float64

This matches the behavior of other binary operations in pandas, like `Series.add()`. To retain the previous behavior, convert the other `Series` to an array before applying the ufunc.

In [54]: np.power(s1, s2.array)
Out[54]:
(continued on next page)
Categorical.argsort now places missing values at the end

Categorical.argsort() now places missing values at the end of the array, making it consistent with NumPy and the rest of pandas (GH21801).

```python
In [55]: cat = pd.Categorical(['b', None, 'a'], categories=['a', 'b'], ordered=True)
```

Previous behavior

```python
In [2]: cat = pd.Categorical(['b', None, 'a'], categories=['a', 'b'], ordered=True)
In [3]: cat.argsort()
Out[3]: array([1, 2, 0])
In [4]: cat[cat.argsort()]
Out[4]: [NaN, a, b]
categories (2, object): [a < b]
```

New behavior

```python
In [56]: cat.argsort()
Out[56]: array([2, 0, 1])
In [57]: cat[cat.argsort()]
Out[57]: ['a', 'b', NaN]
categories (2, object): ['a' < 'b']
```

Column order is preserved when passing a list of dicts to DataFrame

Starting with Python 3.7 the key-order of dict is guaranteed. In practice, this has been true since Python 3.6. The DataFrame constructor now treats a list of dicts in the same way as it does a list of OrderedDict, i.e. preserving the order of the dicts. This change applies only when pandas is running on Python>=3.6 (GH27309).

```python
In [58]: data = [
    ....:     {'name': 'Joe', 'state': 'NY', 'age': 18},
    ....:     {'name': 'Jane', 'state': 'KY', 'age': 19, 'hobby': 'Minecraft'},
    ....:     {'name': 'Jean', 'state': 'OK', 'age': 20, 'finances': 'good'}
    ....: ]
```

Previous Behavior:
The columns were lexicographically sorted previously.
In [1]: `pd.DataFrame(data)`
Out[1]:
<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>finances</th>
<th>hobby</th>
<th>name</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>NaN</td>
<td>NaN</td>
<td>Joe</td>
<td>NY</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>NaN</td>
<td>Minecraft</td>
<td>Jane</td>
<td>KY</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>good</td>
<td>NaN</td>
<td>Jean</td>
<td>OK</td>
</tr>
</tbody>
</table>

**New Behavior:**

The column order now matches the insertion-order of the keys in the dict, considering all the records from top to bottom. As a consequence, the column order of the resulting DataFrame has changed compared to previous pandas versions.

In [59]: `pd.DataFrame(data)`
Out[59]:
<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>state</th>
<th>age</th>
<th>hobby</th>
<th>finances</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Joe</td>
<td>NY</td>
<td>18</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>Jane</td>
<td>KY</td>
<td>19</td>
<td>Minecraft</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>Jean</td>
<td>OK</td>
<td>20</td>
<td>NaN</td>
<td>good</td>
</tr>
</tbody>
</table>

[3 rows x 5 columns]

**Increased minimum versions for dependencies**

Due to dropping support for Python 2.7, a number of optional dependencies have updated minimum versions (GH25725, GH24942, GH25752). Independently, some minimum supported versions of dependencies were updated (GH23519, GH25554). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>1.13.3</td>
<td>X</td>
</tr>
<tr>
<td>pytz</td>
<td>2015.4</td>
<td>X</td>
</tr>
<tr>
<td>python-dateutil</td>
<td>2.6.1</td>
<td>X</td>
</tr>
<tr>
<td>bottleneck</td>
<td>1.2.1</td>
<td></td>
</tr>
<tr>
<td>numexpr</td>
<td>2.6.2</td>
<td></td>
</tr>
<tr>
<td>pytest (dev)</td>
<td>4.0.2</td>
<td></td>
</tr>
</tbody>
</table>

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of pandas. Optional libraries below the lowest tested version may still work, but are not considered supported.
pandas: powerful Python data analysis toolkit, Release 1.4.2

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>beautifulsoup4</td>
<td>4.6.0</td>
</tr>
<tr>
<td>fastparquet</td>
<td>0.2.1</td>
</tr>
<tr>
<td>gcsfs</td>
<td>0.2.2</td>
</tr>
<tr>
<td>lxml</td>
<td>3.8.0</td>
</tr>
<tr>
<td>matplotlib</td>
<td>2.2.2</td>
</tr>
<tr>
<td>openpyxl</td>
<td>2.4.8</td>
</tr>
<tr>
<td>pymysql</td>
<td>0.7.1</td>
</tr>
<tr>
<td>pytables</td>
<td>3.4.2</td>
</tr>
<tr>
<td>scipy</td>
<td>0.19.0</td>
</tr>
<tr>
<td>sqlalchemy</td>
<td>1.1.4</td>
</tr>
<tr>
<td>xarray</td>
<td>0.8.2</td>
</tr>
<tr>
<td>xlrd</td>
<td>1.1.0</td>
</tr>
<tr>
<td>xlsxwriter</td>
<td>0.9.8</td>
</tr>
<tr>
<td>xlwt</td>
<td>1.2.0</td>
</tr>
</tbody>
</table>

See Dependencies and Optional dependencies for more.

Other API changes

- `DatetimeTZDtype` will now standardize pytz timezones to a common timezone instance (GH24713)
- `Timestamp` and `Timedelta` scalars now implement the `to_numpy()` method as aliases to `Timestamp.to_datetime64()` and `Timedelta.to_timedelta64()`, respectively. (GH24653)
- `Timestamp.strptime()` will now raise a `NotImplementedError` (GH25016)
- Comparing `Timestamp` with unsupported objects now returns `NotImplemented` instead of raising `TypeError`. This implies that unsupported rich comparisons are delegated to the other object, and are now consistent with Python 3 behavior for `datetime` objects (GH24011)
- Bug in `DatetimeIndex.snap()` which didn’t preserving the name of the input `Index` (GH25575)
- The `arg` argument in `pandas.core.groupby.DataFrameGroupBy.agg()` has been renamed to `func` (GH26089)
- The `arg` argument in `pandas.core.window._Window.aggregate()` has been renamed to `func` (GH26372)
- Most pandas classes had a `__bytes__` method, which was used for getting a python2-style bytestring representation of the object. This method has been removed as a part of dropping Python2 (GH26447)
- The `.str`-accessor has been disabled for 1-level `MultiIndex`, use `MultiIndex.to_flat_index()` if necessary (GH23679)
- Removed support of gtk package for clipboards (GH26563)
- Using an unsupported version of Beautiful Soup 4 will now raise an `ImportError` instead of a `ValueError` (GH27063)
- `Series.to_excel()` and `DataFrame.to_excel()` will now raise a `ValueError` when saving timezone aware data. (GH27008, GH7056)
- `ExtensionArray.argsort()` places NA values at the end of the sorted array. (GH21801)
- `DataFrame.to_hdf()` and `Series.to_hdf()` will now raise a `NotImplementedError` when saving a `MultiIndex` with extension data types for a fixed format. (GH7775)
• Passing duplicate names in `read_csv()` will now raise a `ValueError` (GH17346)

**Deprecations**

**Sparse subclasses**

The SparseSeries and SparseDataFrame subclasses are deprecated. Their functionality is better-provided by a Series or DataFrame with sparse values.

**Previous way**

```python
df = pd.SparseDataFrame({'A': [0, 0, 1, 2]})
df.dtypes
```

**New way**

```python
In [60]: df = pd.DataFrame({'A': pd.SparseArray([0, 0, 1, 2])})
In [61]: df.dtypes
Out[61]:
A    Sparse[int64, 0]
Length: 1, dtype: object
```

The memory usage of the two approaches is identical. See *Migrating* for more (GH19239).

**msgpack format**

The msgpack format is deprecated as of 0.25 and will be removed in a future version. It is recommended to use pyarrow for on-the-wire transmission of pandas objects. (GH27084)

**Other deprecations**

• The deprecated `.ix[]` indexer now raises a more visible `FutureWarning` instead of `DeprecationWarning` (GH26438).

• Deprecated the `units=M` (months) and `units=Y` (year) parameters for `units` of `pandas.to_timedelta()`, `pandas.Timedelta()` and `pandas.TimedeltaIndex()` (GH16344)

• `pandas.concat()` has deprecated the `join_axes`-keyword. Instead, use `DataFrame.reindex()` or `DataFrame.reindex_like()` on the result or on the inputs (GH21951)

• The `SparseArray.values` attribute is deprecated. You can use `np.asarray(...)` or the `SparseArray.to_dense()` method instead (GH26421).

• The functions `pandas.to_datetime()` and `pandas.to_timedelta()` have deprecated the `box` keyword. Instead, use `to_numpy()` or `Timestamp.to_datetime64()` or `Timedelta.to_timedelta64()` (GH24416)

• The `DataFrame.compound()` and `Series.compound()` methods are deprecated and will be removed in a future version (GH26405).

• The internal attributes `_start`, `_stop` and `_step` attributes of `RangeIndex` have been deprecated. Use the public attributes `start`, `stop` and `step` instead (GH26581).

• The `Series.ftype()`, `Series.ftypes()` and `DataFrame.ftypes()` methods are deprecated and will be removed in a future version. Instead, use `Series.dtype()` and `DataFrame.dtypes()` (GH26705).
• The Series.get_values(), DataFrame.get_values(), Index.get_values(), SparseArray.get_values() and Categorical.get_values() methods are deprecated. One of np.asarray() or to_numpy() can be used instead (GH19617).

• The ‘outer’ method on NumPy ufuncs, e.g. np.subtract.outer has been deprecated on Series objects. Convert the input to an array with Series.array first (GH27186)

• Timedelta.resolution() is deprecated and replaced with Timedelta.resolution_string(). In a future version, Timedelta.resolution() will be changed to behave like the standard library datetime.timedelta.resolution (GH21344)

• read_table() has been undeprecated. (GH25220)

• Index.dtype_str is deprecated. (GH18262)

• Series.imag and Series.real are deprecated. (GH18262)

• Series.put() is deprecated. (GH18262)

• Index.item() and Series.item() is deprecated. (GH18262)

• The default value ordered=None in CategoricalDtype has been deprecated in favor of ordered=False. When converting between categorical types ordered=True must be explicitly passed in order to be preserved. (GH26336)

• Index.contains() is deprecated. Use key in index(__contains__) instead (GH17753).

• DataFrame.get_dtype_counts() is deprecated. (GH18262)

• Categorical.ravel() will return a Categorical instead of a np.ndarray (GH27199)

Removal of prior version deprecations/changes

• Removed Panel (GH25047, GH25191, GH25231)

• Removed the previously deprecated sheetname keyword in read_excel() (GH16442, GH20938)

• Removed the previously deprecated TimeGrouper (GH16942)

• Removed the previously deprecated parse_cols keyword in read_excel() (GH16488)

• Removed the previously deprecated pd.options.html.border (GH16970)

• Removed the previously deprecated convert_objects (GH11221)

• Removed the previously deprecated select method of DataFrame and Series (GH17633)

• Removed the previously deprecated behavior of Series treated as list-like in rename_categories() (GH17982)

• Removed the previously deprecated DataFrame.reindex_axis and Series.reindex_axis (GH17842)

• Removed the previously deprecated behavior of altering column or index labels with Series.rename_axis() or DataFrame.rename_axis() (GH17842)

• Removed the previously deprecated tupleize_cols keyword argument in read_html(), read_csv(), and DataFrame.to_csv() (GH17877, GH17820)

• Removed the previously deprecated DataFrame.from_csv and Series.from_csv (GH17812)

• Removed the previously deprecated raise_on_error keyword argument in DataFrame.where() and DataFrame.mask() (GH17744)

• Removed the previously deprecated ordered and categories keyword arguments in astype (GH17742)

• Removed the previously deprecated cdate_range (GH17691)
- Removed the previously deprecated `True` option for the `dropna` keyword argument in `SeriesGroupBy.nth()` (GH17493)
- Removed the previously deprecated `convert` keyword argument in `Series.take()` and `DataFrame.take()` (GH17352)
- Removed the previously deprecated behavior of arithmetic operations with `datetime.date` objects (GH21152)

**Performance improvements**

- Significant speedup in `SparseArray` initialization that benefits most operations, fixing performance regression introduced in v0.20.0 (GH24985)
- `DataFrame.to_stata()` is now faster when outputting data with any string or non-native endian columns (GH25045)
- Improved performance of `Series.searchsorted()`. The speedup is especially large when the dtype is `int8/int16/int32` and the searched key is within the integer bounds for the dtype (GH22034)
- Improved performance of `pandas.core.groupby.GroupBy.quantile()` (GH20405)
- Improved performance of slicing and other selected operation on a `RangeIndex` (GH26565, GH26617, GH26722)
- `RangeIndex` now performs standard lookup without instantiating an actual hashtable, hence saving memory (GH16685)
- Improved performance of `read_csv()` by faster tokenizing and faster parsing of small float numbers (GH25784)
- Improved performance of `read_csv()` by faster parsing of N/A and boolean values (GH25804)
- Improved performance of `IntervalIndex.is_monotonic`, `IntervalIndex.is_monotonic_increasing` and `IntervalIndex.is_monotonic_decreasing` by removing conversion to `MultiIndex` (GH24813)
- Improved performance of `DataFrame.to_csv()` when writing datetime dtypes (GH25708)
- Improved performance of `read_csv()` by much faster parsing of `MM/YYYY` and `DD/MM/YYYY` datetime formats (GH25922)
- Improved performance of `IntervalIndex.is_unique` by removing conversion to `MultiIndex` (GH24813)
- Improved performance by removing the need for a garbage collect when checking for `SettingWithCopyWarning` (GH27031)
- For `to_datetime()` changed default value of cache parameter to `True` (GH26043)
- Improved performance of `DatetimeIndex` and `PeriodIndex` slicing given non-unique, monotonic data (GH27136)
- Improved performance of `pd.read_json()` for index-oriented data. (GH26773)
• Improved performance of `MultiIndex.shape()` (GH27384).

Bug fixes

Categorical

• Bug in `DataFrame.at()` and `Series.at()` that would raise exception if the index was a `CategoricalIndex` (GH20629)
• Fixed bug in comparison of ordered `Categorical` that contained missing values with a scalar which sometimes incorrectly resulted in `True` (GH26504)
• Bug in `DataFrame.dropna()` when the `DataFrame` has a `CategoricalIndex` containing `Interval` objects incorrectly raised a `TypeError` (GH25087)

Datetimelike

• Bug in `to_datetime()` which would raise an (incorrect) `ValueError` when called with a date far into the future and the `format` argument specified instead of raising `OutOfBoundsDatetime` (GH23830)
• Bug in `to_datetime()` which would raise `InvalidIndexError: Reindexing only valid with uniquely valued Index objects when called with cache=True, with arg including at least two different elements from the set {None, numpy.nan, pandas.NaT}` (GH22305)
• Bug in `DataFrame` and `Series` where timezone aware data with `dtype='datetime64[ns]` was not cast to naive (GH25843)
• Improved `Timestamp` type checking in various datetime functions to prevent exceptions when using a subclassed `datetime` (GH25851)
• Bug in `Series` and `DataFrame` repr where `np.datetime64('NaT')` and `np.timedelta64('NaT')` with `dtype=object` would be represented as `NaN` (GH25445)
• Bug in `to_datetime()` which does not replace the invalid argument with `NaT` when error is set to coerce (GH26122)
• Bug in adding `DateOffset` with nonzero month to `Datetimedelta` would raise `ValueError` (GH26258)
• Bug in `to_datetime()` which raises unhandled `OverflowError` when called with mix of invalid dates and NaN values with `format='%Y%m%d' and error='coerce'` (GH25512)
• Bug in `isin()` for datetimelike indexes; `Datetimedelta`, `TimedeltaIndex` and `PeriodIndex` where the `levels` parameter was ignored. (GH26675)
• Bug in `to_datetime()` which raises `TypeError` for `format='%Y%m%d' when called for invalid integer dates with length >= 6 digits with errors='ignore'`
• Bug when comparing a `PeriodIndex` against a zero-dimensional numpy array (GH26689)
• Bug in constructing a `Series` or `DataFrame` from a `numpy.datetime64` array with a non-ns unit and out-of-bound timestamps generating rubbish data, which will now correctly raise an `OutOfBoundsDatetime` error (GH26206).
• Bug in `date_range()` with unnecessary `OverflowError` being raised for very large or very small dates (GH26651)
• Bug where adding `Timestamp` to a `np.timedelta64` object would raise instead of returning a `Timestamp` (GH24775)
Bug where comparing a zero-dimensional numpy array containing a `np.datetime64` object to a `Timestamp` would incorrect raise `TypeError` (GH26916)

Bug in `to_datetime()` which would raise `ValueError`: Tz-aware datetime.datetime cannot be converted to datetime64 unless utc=True when called with cache=True, with arg including datetime strings with different offset (GH26097)

### Timedelta

Bug in `TimedeltaIndex.intersection()` where for non-monotonic indices in some cases an empty Index was returned when in fact an intersection existed (GH25913)

Bug with comparisons between `Timedelta` and NaT raising `TypeError` (GH26039)

Bug when adding or subtracting a `BusinessHour` to a `Timestamp` with the resulting time landing in a following or prior day respectively (GH26381)

Bug when comparing a `TimedeltaIndex` against a zero-dimensional numpy array (GH26689)

### Timezones

Bug in `DatetimeIndex.to_frame()` where timezone aware data would be converted to timezone naive data (GH25809)

Bug in `to_datetime()` with utc=True and datetime strings that would apply previously parsed UTC offsets to subsequent arguments (GH24992)

Bug in `Timestamp.tz_localize()` and `Timestamp.tz_convert()` does not propagate `freq` (GH25241)

Bug in `Series.at()` where setting `Timestamp` with timezone raises `TypeError` (GH25506)

Bug in `DataFrame.update()` when updating with timezone aware data would return timezone naive data (GH25807)

Bug in `to_datetime()` where an uninformative `RuntimeError` was raised when passing a naive `Timestamp` with datetime strings with mixed UTC offsets (GH25978)

Bug in `to_datetime()` with unit='ns' would drop timezone information from the parsed argument (GH26168)

Bug in `DataFrame.join()` where joining a timezone aware index with a timezone aware column would result in a column of NaN (GH26335)

Bug in `date_range()` where ambiguous or nonexistent start or end times were not handled by the ambiguous or nonexistent keywords respectively (GH27088)

Bug in `DatetimeIndex.union()` when combining a timezone aware and timezone unaware `DatetimeIndex` (GH21671)

Bug when applying a numpy reduction function (e.g. `numpy.minimum()`) to a timezone aware `Series` (GH15552)
Numeric

- Bug in to_numeric() in which large negative numbers were being improperly handled (GH24910)
- Bug in to_numeric() in which numbers were being coerced to float, even though errors was not coerce (GH24910)
- Bug in to_numeric() in which invalid values for errors were being allowed (GH26466)
- Bug in format in which floating point complex numbers were not being formatted to proper display precision and trimming (GH25514)
- Bug in error messages in DataFrame.corr() and Series.corr(). Added the possibility of using a callable. (GH25729)
- Bug in Series.divmod() and Series.rdivmod() which would raise an (incorrect) ValueError rather than return a pair of Series objects as result (GH25557)
- Raises a helpful exception when a non-numeric index is sent to interpolate() with methods which require numeric index. (GH21662)
- Bug in eval() when comparing floats with scalar operators, for example: x < -0.1 (GH25928)
- Fixed bug where casting all-boolean array to integer extension array failed (GH25211)
- Bug in divmod with a Series object containing zeros incorrectly raising AttributeError (GH26987)
- Inconsistency in Series floor-division (//) and divmod filling positive//zero with NaN instead of Inf (GH27321)

Conversion

- Bug in DataFrame.astype() when passing a dict of columns and types the errors parameter was ignored. (GH25905)

Strings

- Bug in the __name__ attribute of several methods of Series.str, which were set incorrectly (GH23551)
- Improved error message when passing Series of wrong dtype to Series.str.cat() (GH22722)

Interval

- Construction of Interval is restricted to numeric, Timestamp and Timedelta endpoints (GH23013)
- Fixed bug in Series/DataFrame not displaying NaN in IntervalIndex with missing values (GH25984)
- Bug in IntervalIndex.get_loc() where a KeyError would be incorrectly raised for a decreasing IntervalIndex (GH25860)
- Bug in Index constructor where passing mixed closed Interval objects would result in a ValueError instead of an object dtype Index (GH27172)
Indexing

- Improved exception message when calling `DataFrame.iloc()` with a list of non-numeric objects (GH25753).
- Improved exception message when calling `.iloc` or `.loc` with a boolean indexer with different length (GH26658).
- Bug in `KeyError` exception message when indexing a `MultiIndex` with a non-existent key not displaying the original key (GH27250).
- Bug in `.iloc` and `.loc` with a boolean indexer not raising an `IndexError` when too few items are passed (GH26658).
- Bug in `DataFrame.loc()` and `Series.loc()` where `KeyError` was not raised for a `MultiIndex` when the key was less than or equal to the number of levels in the `MultiIndex` (GH14885).
- Bug in which `DataFrame.append()` produced an erroneous warning indicating that a `KeyError` will be thrown in the future when the data to be appended contains new columns (GH22252).
- Bug in which `DataFrame.to_csv()` caused a segfault for a reindexed data frame, when the indices were single-level `MultiIndex` (GH26303).
- Fixed bug where assigning a `arrays.PandasArray` to a `pandas.core.frame.DataFrame` would raise error (GH26390)
- Allow keyword arguments for callable local reference used in the `DataFrame.query()` string (GH26426)
- Fixed a `KeyError` when indexing a `MultiIndex` level with a list containing exactly one label, which is missing (GH27148)
- Bug which produced `AttributeError` on partial matching `Timestamp` in a `MultiIndex` (GH26944)
- Bug in `Categorical` and `CategoricalIndex` with `Interval` values when using the in operator (`__contains__`) with objects that are not comparable to the values in the `Interval` (GH23705)
- Bug in `DataFrame.loc()` and `DataFrame.iloc()` on a `DataFrame` with a single timezone-aware date-time64[ns] column incorrectly returning a scalar instead of a `Series` (GH27110)
- Bug in `CategoricalIndex` and `Categorical` incorrectly raising `ValueError` instead of `TypeError` when a list is passed using the in operator (`__contains__`) (GH21729)
- Bug in setting a new value in a `Series` with a `Timedelta` object incorrectly casting the value to an integer (GH22717)
- Bug in `Series` setting a new key (`__setitem__`) with a timezone-aware datetime incorrectly raising `ValueError` (GH12862)
- Bug in `DataFrame.iloc()` when indexing with a read-only indexer (GH17192)
- Bug in `Series` setting an existing tuple key (`__setitem__`) with timezone-aware datetime values incorrectly raising `TypeError` (GH20441)
Missing

- Fixed misleading exception message in `Series.interpolate()` if argument `order` is required, but omitted (GH10633, GH24014).
- Fixed class type displayed in exception message in `DataFrame.dropna()` if invalid `axis` parameter passed (GH25555).
- A `ValueError` will now be thrown by `DataFrame.fillna()` when `limit` is not a positive integer (GH27042).

MultiIndex

- Bug in which incorrect exception raised by `Timedelta` when testing the membership of `MultiIndex` (GH24570).

IO

- Bug in `DataFrame.to_html()` where values were truncated using display options instead of outputting the full content (GH17004).
- Fixed bug in missing text when using `to_clipboard()` if copying utf-16 characters in Python 3 on Windows (GH25040).
- Bug in `read_json()` for `orient='table'` when it tries to infer dtypes by default, which is not applicable as dtypes are already defined in the JSON schema (GH21345).
- Bug in `read_json()` for `orient='table'` and float index, as it infers index dtype by default, which is not applicable because index dtype is already defined in the JSON schema (GH25433).
- Bug in `read_json()` for `orient='table'` and string of float column names, as it makes a column name type conversion to `Timestamp`, which is not applicable because column names are already defined in the JSON schema (GH25435).
- Bug in `json_normalize()` for `errors='ignore'` where missing values in the input data, were filled in resulting `DataFrame` with the string "nan" instead of `numpy.nan` (GH25468).
- `DataFrame.to_html()` now raises `TypeError` when using an invalid type for the `classes` parameter instead of `AssertionError` (GH25608).
- Bug in `DataFrame.to_string()` and `DataFrame.to_latex()` that would lead to incorrect output when the `header` keyword is used (GH16718).
- Bug in `read_csv()` not properly interpreting the UTF8 encoded filenames on Windows on Python 3.6+ (GH15086).
- Improved performance in `pandas.read_stata()` and `pandas.io.stata.StataReader` when converting columns that have missing values (GH25772).
- Bug in `DataFrame.to_html()` where header numbers would ignore display options when rounding (GH17280).
- Bug in `read_hdf()` where reading a table from an HDF5 file written directly with PyTables fails with a `ValueError` when using a sub-selection via the `start` or `stop` arguments (GH11188).
- Bug in `read_hdf()` not properly closing store after a `KeyError` is raised (GH25766).
- Improved the explanation for the failure when value labels are repeated in Stata dta files and suggested workarounds (GH25772).
• Improved `pandas.read_stata()` and `pandas.io.stata.StataReader` to read incorrectly formatted 118 format files saved by Stata (GH25960)

• Improved the `col_space` parameter in `DataFrame.to_html()` to accept a string so CSS length values can be set correctly (GH25941)

• Fixed bug in loading objects from S3 that contain # characters in the URL (GH25945)

• Adds `use_bqstorage_api` parameter to `read_gbq()` to speed up downloads of large data frames. This feature requires version 0.10.0 of the pandas-gbq library as well as the `google-cloud-bigquery-storage` and `fastavro` libraries. (GH26104)

• Fixed memory leak in `DataFrame.to_json()` when dealing with numeric data (GH24889)

• Bug in `read_json()` where date strings with Z were not converted to a UTC timezone (GH26168)

• Added `cache_dates=True` parameter to `read_csv()`, which allows to cache unique dates when they are parsed (GH25990)

• `DataFrame.to_excel()` now raises a `ValueError` when the caller’s dimensions exceed the limitations of Excel (GH26051)

• Fixed bug in `pandas.read_csv()` where a BOM would result in incorrect parsing using engine='python' (GH26545)

• `read_excel()` now raises a `ValueError` when input is of type `pandas.io.excel.ExcelFile` and engine param is passed since `pandas.io.excel.ExcelFile` has an engine defined (GH26566)

• Bug while selecting from HDFStore with where='' specified (GH26610)

• Fixed bug in `DataFrame.to_excel()` where custom objects (i.e. `PeriodIndex`) inside merged cells were not being converted into types safe for the Excel writer (GH27006)

• Bug in `read_hdf()` where reading a timezone aware `DatetimeIndex` would raise a `TypeError` (GH11926)

• Bug in `to_msgpack()` and `read_msgpack()` which would raise a `ValueError` rather than a `FileNotFoundError` for an invalid path (GH27160)

• Fixed bug in `DataFrame.to_parquet()` which would raise a `ValueError` when the dataframe had no columns (GH27339)

• Allow parsing of `PeriodDType` columns when using `read_csv()` (GH26934)

**Plotting**

• Fixed bug where `api.extensions.ExtensionArray` could not be used in matplotlib plotting (GH25587)

• Bug in an error message in `DataFrame.plot()`. Improved the error message if non-numerics are passed to `DataFrame.plot()` (GH25481)

• Bug in incorrect ticklabel positions when plotting an index that are non-numeric / non-datetime (GH7612, GH15912, GH22334)

• Fixed bug causing plots of `PeriodIndex` timeseries to fail if the frequency is a multiple of the frequency rule code (GH14763)

• Fixed bug when plotting a `DatetimeIndex` with `datetime.timezone.utc` timezone (GH17173)
**GroupBy/resample/rolling**

- Bug in `pandas.core.resample.Resampler.agg()` with a timezone aware index where `OverflowError` would raise when passing a list of functions (GH22660)

- Bug in `pandas.core.groupby.DataFrameGroupBy.nunique()` in which the names of column levels were lost (GH23222)

- Bug in `pandas.core.groupby.GroupBy.agg()` when applying an aggregation function to timezone aware data (GH23683)

- Bug in `pandas.core.groupby.GroupBy.first()` and `pandas.core.groupby.GroupBy.last()` where timezone information would be dropped (GH21603)

- Bug in `pandas.core.groupby.GroupBy.size()` when grouping only NA values (GH23050)

- Bug in `Series.groupby()` where observed kwarg was previously ignored (GH24880)

- Bug in `Series.groupby()` where using `groupby` with a `MultiIndex` Series with a list of labels equal to the length of the series caused incorrect grouping (GH25704)

- Ensured that ordering of outputs in `groupby` aggregation functions is consistent across all versions of Python (GH25692)

- Ensured that result group order is correct when grouping on an ordered Categorical and specifying `observed=True` (GH25871, GH25167)

- Bug in `pandas.core.window.Rolling.min()` and `pandas.core.window.Rolling.max()` that caused a memory leak (GH25893)

- Bug in `pandas.core.window.Rolling.count()` and `pandas.core.window.Expanding.count` was previously ignoring the `axis` keyword (GH13503)

- Bug in `pandas.core.groupby.GroupBy.idxmax()` and `pandas.core.groupby.GroupBy.idxmin()` with datetime column would return incorrect dtype (GH25444, GH15306)

- Bug in `pandas.core.groupby.GroupBy.cumsum()`, `pandas.core.groupby.GroupBy.cumprod()`, `pandas.core.groupby.GroupBy.cummin()` and `pandas.core.groupby.GroupBy.cummax()` with categorical column having absent categories, would return incorrect result or segfault (GH16771)

- Bug in `pandas.core.groupby.GroupBy.nth()` where NA values in the grouping would return incorrect results (GH26011)

- Bug in `pandas.core.groupby.SeriesGroupBy.transform()` where transforming an empty group would raise a `ValueError` (GH26208)

- Bug in `pandas.core.frame.DataFrame.groupby()` where passing a `pandas.core.groupby.grouper.Grouper` would return incorrect groups when using the `.groups` accessor (GH26326)

- Bug in `pandas.core.groupby.GroupBy.agg()` where incorrect results are returned for uint64 columns. (GH26310)

- Bug in `pandas.core.window.Rolling.median()` and `pandas.core.window.Rolling.quantile()` where MemoryError is raised with empty window (GH26005)

- Bug in `pandas.core.window.Rolling.median()` and `pandas.core.window.Rolling.quantile()` where incorrect results are returned with `closed='left'` and `closed='neither'` (GH26005)

- Improved `pandas.core.window.Rolling`, `pandas.core.window.Window` and `pandas.core.window. ExponentialMovingWindow` functions to exclude nuisance columns from results instead of raising errors and raise a `DataError` only if all columns are nuisance (GH12537)

- Bug in `pandas.core.window.Rolling.max()` and `pandas.core.window.Rolling.min()` where incorrect results are returned with an empty variable window (GH26005)
• Raise a helpful exception when an unsupported weighted window function is used as an argument of pandas.core.window.Window.aggregate() (GH26597)

Reshaping

• Bug in pandas.merge() adds a string of None, if None is assigned in suffixes instead of remain the column name as-is (GH24782).
• Bug in merge() when merging by index name would sometimes result in an incorrectly numbered index (missing index values are now assigned NA) (GH24212, GH25009)
• to_records() now accepts dtypes to its column_dtypes parameter (GH24895)
• Bug in concat() where order of OrderedDict (and dict in Python 3.6+) is not respected, when passed in as objs argument (GH21510)
• Bug in pivot_table() where columns with NaN values are dropped even if dropna argument is False, when the aggfunc argument contains a list (GH22159)
• Bug in concat() where the resulting freq of two DatetimeIndex with the same freq would be dropped (GH3232).
• Bug in merge() where merging with equivalent Categorical dtypes was raising an error (GH22501)
• Bug in DataFrame instantiating with a dict of iterators or generators (e.g. pd.DataFrame({'A': reversed(range(3)))}) raised an error (GH26349).
• Bug in DataFrame instantiating with a range (e.g. pd.DataFrame(range(3))) raised an error (GH26342).
• Bug in DataFrame constructor when passing non-empty tuples would cause a segmentation fault (GH25691)
• Bug in Series.apply() failed when the series is a timezone aware DatetimeIndex (GH25959)
• Bug in pandas.cut() where large bins could incorrectly raise an error due to an integer overflow (GH26045)
• Bug in DataFrame.sort_index() where an error is thrown when a multi-indexed DataFrame is sorted on all levels with the initial level sorted last (GH26053)
• Bug in Series.nlargest() treats True as smaller than False (GH26154)
• Bug in DataFrame.pivot_table() with a IntervalIndex as pivot index would raise TypeError (GH25814)
• Bug in which DataFrame.from_dict() ignored order of OrderedDict when orient='index' (GH8425).
• Bug in DataFrame.transpose() where transposing a DataFrame with a timezone-aware datetime column would incorrectly raise ValueError (GH26825)
• Bug in pivot_table() when pivoting a timezone aware column as the values would remove timezone information (GH14948)
• Bug in merge_asof() when specifying multiple by columns where one is datetime64[ns, tz] dtype (GH26649)
Sparse

- Significant speedup in `SparseArray` initialization that benefits most operations, fixing performance regression introduced in v0.20.0 (GH24985)
- Bug in `SparseFrame` constructor where passing `None` as the data would cause `default_fill_value` to be ignored (GH16807)
- Bug in `SparseDataFrame` when adding a column in which the length of values does not match length of index, `AssertionError` is raised instead of raising `ValueError` (GH25484)
- Introduce a better error message in `Series.sparse.from_coo()` so it returns a `TypeError` for inputs that are not coo matrices (GH26554)
- Bug in `numpy.modf()` on a `SparseArray`. Now a tuple of `SparseArray` is returned (GH26946).

Build changes

- Fix install error with PyPy on macOS (GH26536)

ExtensionArray

- Bug in `factorize()` when passing an `ExtensionArray` with a custom `na_sentinel` (GH25696).
- `Series.count()` miscounts NA values in ExtensionArrays (GH26835)
- Added `Series.__array_ufunc__` to better handle NumPy ufuncs applied to Series backed by extension arrays (GH23293).
- Keyword argument `deep` has been removed from `ExtensionArray.copy()` (GH27083)

Other

- Removed unused C functions from vendored UltraJSON implementation (GH26198)
- Allow `Index` and `RangeIndex` to be passed to `numpy min` and `max` functions (GH26125)
- Use actual class name in repr of empty objects of a `Series` subclass (GH27001).
- Bug in `DataFrame` where passing an object array of timezone-aware `datetime` objects would incorrectly raise `ValueError` (GH13287)

Contributors

A total of 231 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- 1_x7 +
- Abdullah İhsan Seçer +
- Adam Bull +
- Adam Hooper
- Albert Villanova del Moral
- Alex Watt +
• AlexTereshenkov +
• Alexander Buchkovsky
• Alexander Hendorf +
• Alexander Nordin +
• Alexander Ponomaroff
• Alexandre Batisse +
• Alexandre Decan +
• Allen Downey +
• Alyssa Fu Ward +
• Andrew Gaspari +
• Andrew Wood +
• Antoine Viscardi +
• Antonio Gutierrez +
• Arno Veenstra +
• ArtinSarraf
• Batalex +
• Baurzhan Muftakhidinov
• Benjamin Rowell
• Bharat Raghunathan +
• Bhavani Ravi +
• Big Head +
• Brett Randall +
• Bryan Cutler +
• C John Klehm +
• Caleb Braun +
• Cecilia +
• Chris Bertinato +
• Chris Stadler +
• Christian Haege +
• Christian Hudon
• Christopher Whelan
• Chuanzhu Xu +
• Clemens Brunner
• Damian Kula +
• Daniel Hrisca +
• Daniel Luis Costa +
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Daniel Saxton
- DanielFEvans +
- David Liu +
- Deepyaman Datta +
- Denis Belavin +
- Devin Petersohn +
- Diane Trout +
- EdAbati +
- Enrico Rotundo +
- EternalLearner42 +
- Evan +
- Evan Livelo +
- Fabian Rost +
- Flavien Lambert +
- Florian Rathgeber +
- Frank Hoang +
- Gaibo Zhang +
- Gioia Ballin
- Giuseppe Romagnuolo +
- Gordon Blackadder +
- Gregory Rome +
- Guillaume Gay
- HHest +
- Hielke Walinga +
- How Si Wei +
- Hubert
- Huize Wang +
- Hyukjin Kwon +
- Ian Dunn +
- Inevitable-Marzipan +
- Irv Lustig
- JElfner +
- Jacob Bundgaard +
- James Cobon-Kerr +
- Jan-Philip Gehrcke +
- Jarrod Millman +
• Jayanth Katuri +
• Jeff Reback
• Jeremy Schendel
• Jiang Yue +
• Joel Ostblom
• Johan von Forstner +
• Johnny Chiu +
• Jonas +
• Jonathon VandeZande +
• Jop Vermeer +
• Joris Van den Bossche
• Josh
• Josh Friedlander +
• Justin Zheng
• Kaiqi Dong
• Kane +
• Kapil Patel +
• Kara de la Marck +
• Katherine Surta +
• Katrin Leinweber +
• Kendall Masse
• Kevin Sheppard
• Kyle Kosic +
• Lorenzo Stella +
• Maarten Rietbergen +
• Mak Sze Chun
• Marc Garcia
• Mateusz Woś
• Matias Heikkilä
• Mats Maiwald +
• Matthew Roeschke
• Max Bolingbroke +
• Max Kovalovs +
• Max van Deursen +
• Michael
• Michael Davis +
• Michael P. Moran +
• Mike Cramblett +
• Min ho Kim +
• Misha Veldhoen +
• Mukul Ashwath Ram +
• MusTheDataGuy +
• Nanda H Krishna +
• Nicholas Musolino
• Noam Hershtig +
• Noora Husseini +
• Paul
• Paul Reidy
• Pauli Virtanen
• Pav A +
• Peter Leimbigler +
• Philippe Ombredanne +
• Pietro Battiston
• Richard Eames +
• Roman Yurchak
• Ruijing Li
• Ryan
• Ryan Joyce +
• Ryan Nazareth
• Ryan Rehman +
• Sakar Panta +
• Samuel Sinayoko
• Sandeep Pathak +
• Sangwoong Yoon
• Saurav Chakravorty
• Scott Talbert +
• Sergey Kopylov +
• Shantanu Gontia +
• Shivam Rana +
• Shorokhov Sergey +
• Simon Hawkins
• Soyoun(Rose) Kim
• Stephan Hoyer
• Stephen Cowley +
• Stephen Rauch
• Sterling Paramore +
• Steven +
• Stijn Van Hoey
• Sumanau Sareen +
• Takuya N +
• Tan Tran +
• Tao He +
• Tarbo Fukazawa
• Terji Petersen +
• Thein Oo
• ThibTrip +
• Thijs Damsma +
• Thiviyan Thanapalasingam
• Thomas A Caswell
• Thomas Kluiters +
• Tilen Kusterle +
• Tim Gates +
• Tim Hoffmann
• Tim Swast
• Tom Augspurger
• Tom Neep +
• Tomáš Chvátal +
• Tyler Reddy
• Vaibhav Vishal +
• Vasily Litvinov +
• Vibhu Agarwal +
• Vikramjeet Das +
• Vladislav +
• Víctor Moron Tejero +
• Wenhuan
• Will Ayd +
• William Ayd
• Wouter De Coster +
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Yoann Goular +
- Zach Angell +
- alimcmaster1
- anmyachev +
- chris-b1
- danielplawrence +
- endenis +
- enisnazif +
- ezcitron +
- fjetter
- froessler
- gfyoun
- gwrome +
- h-vetinari
- haison +
- hannah-c +
- heckeop +
- iamshwin +
- jamesoliverh +
- jbrockmendel
- jkovacevic +
- killerontherun1 +
- knuu +
- kpapdac +
- kpflugshaupt +
- krsnik93 +
- leerssej +
- lrjball +
- mazayo +
- nathalier +
- nreben +
- nullptr +
- pilkibun +
- pmaxey83 +
- rbenes +
- robbuckley
5.7 Version 0.24

5.7.1 What's new in 0.24.2 (March 12, 2019)

**Warning:** The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See Dropping Python 2.7 for more.

These are the changes in pandas 0.24.2. See Release notes for a full changelog including other versions of pandas.

**Fixed regressions**

- Fixed regression in `DataFrame.all()` and `DataFrame.any()` where bool_only=True was ignored (GH25101)
- Fixed issue in DataFrame construction with passing a mixed list of mixed types could segfault. (GH25075)
- Fixed regression in `DataFrame.apply()` causing RecursionError when dict-like classes were passed as argument. (GH25196)
- Fixed regression in `DataFrame.replace()` where regex=True was only replacing patterns matching the start of the string (GH25259)
- Fixed regression in `DataFrame.duplicated()`, where empty dataframe was not returning a boolean dtyped Series. (GH25184)
- Fixed regression in `Series.min()` and `Series.max()` where numeric_only=True was ignored when the Series contained Categorical data (GH25299)
- Fixed regression in subtraction between Series objects with datetime64[ns] dtype incorrectly raising OverflowError when the Series on the right contains null values (GH25317)
- Fixed regression in `TimedeltaIndex` where np.sum(index) incorrectly returned a zero-dimensional object instead of a scalar (GH25282)
- Fixed regression in `IntervalDtype` construction where passing an incorrect string with ‘Interval’ as a prefix could result in a RecursionError. (GH25338)
- Fixed regression in creating a period-dtype array from a read-only NumPy array of period objects. (GH25403)
• Fixed regression in `Categorical`, where constructing it from a categorical `Series` and an explicit `categories` that differed from that in the `Series` created an invalid object which could trigger segfaults. (GH25318)

• Fixed regression in `to_timedelta()` losing precision when converting floating data to `Timedelta` data (GH25077).

• Fixed pip installing from source into an environment without NumPy (GH25193)

• Fixed regression in `DataFrame.replace()` where large strings of numbers would be coerced into `int64`, causing an `OverflowError` (GH25616)

• Fixed regression in `factorize()` when passing a custom `na_sentinel` value with `sort=True` (GH25409).

• Fixed regression in `DataFrame.to_csv()` writing duplicate line endings with gzip compress (GH25311)

**Bug fixes**

**I/O**

• Better handling of terminal printing when the terminal dimensions are not known (GH25080)

• Bug in reading a HDF5 table-format `DataFrame` created in Python 2, in Python 3 (GH24925)

• Bug in reading a JSON with `orient='table'` generated by `DataFrame.to_json()` with `index=False` (GH25170)

• Bug where float indexes could have misaligned values when printing (GH25061)

**Categorical**

• Bug where calling `Series.replace()` on categorical data could return a `Series` with incorrect dimensions (GH24971)

•

**Reshaping**

• Bug in `transform()` where applying a function to a timezone aware column would return a timezone naive result (GH24198)

• Bug in `DataFrame.join()` when joining on a timezone aware `DatetimeIndex` (GH23931)

**Visualization**

• Bug in `Series.plot()` where a secondary y axis could not be set to log scale (GH25545)

**Other**

• Bug in `Series.is_unique()` where single occurrences of NaN were not considered unique (GH25180)

• Bug in `merge()` when merging an empty `DataFrame` with an `Int64` column or a non-empty `DataFrame` with an `Int64` column that is all NaN (GH25183)

• Bug in `IntervalTree` where a `RecursionError` occurs upon construction due to an overflow when adding endpoints, which also causes `IntervalIndex` to crash during indexing operations (GH25485)

• Bug in `Series.size` raising for some extension-array-backed `Series`, rather than returning the size (GH25580)

• Bug in resampling raising for nullable integer-dtype columns (GH25580)
Contributors

A total of 25 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Albert Villanova del Moral
- Arno Veenstra +
- chris-b1
- Devin Petersohn +
- EternalLearner42 +
- Flavien Lambert +
- gyoung
- Gioia Ballin
- jbrockmendel
- Jeff Reback
- Jeremy Schendel
- Johan von Forstner +
- Joris Van den Bossche
- Josh
- Justin Zheng
- Kendall Masse
- Matthew Roeschke
- Max Bolingbroke +
- rbenes +
- Sterling Paramore +
- Tao He +
- Thomas A Caswell
- Tom Augspurger
- Vibhu Agarwal +
- William Ayd
- Zach Angell
5.7.2 What’s new in 0.24.1 (February 3, 2019)

**Warning:** The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See [Dropping Python 2.7](#) for more.

These are the changes in pandas 0.24.1. See [Release notes](#) for a full changelog including other versions of pandas. See [What’s new in 0.24.0 (January 25, 2019)](#) for the 0.24.0 changelog.

### API changes

**Changing the sort parameter for Index set operations**

The default sort value for `Index.union()` has changed from `True` to `None` (GH24959). The default behavior, however, remains the same: the result is sorted, unless

1. `self` and `other` are identical
2. `self` or `other` is empty
3. `self` or `other` contain values that can not be compared (a `RuntimeWarning` is raised).

This change will allow `sort=True` to mean “always sort” in a future release.

The same change applies to `Index.difference()` and `Index.symmetric_difference()`, which would not sort the result when the values could not be compared.

The `sort` option for `Index.intersection()` has changed in three ways.

1. The default has changed from `True` to `False`, to restore the pandas 0.23.4 and earlier behavior of not sorting by default.
2. The behavior of `sort=True` can now be obtained with `sort=None`. This will sort the result only if the values in `self` and `other` are not identical.
3. The value `sort=True` is no longer allowed. A future version of pandas will properly support `sort=True` meaning “always sort”.

### Fixed regressions

- Fixed regression in `DataFrame.to_dict()` with `records` orient raising an `AttributeError` when the `DataFrame` contained more than 255 columns, or wrongly converting column names that were not valid python identifiers (GH24939, GH24940).
- Fixed regression in `read_sql()` when passing certain queries with MySQL/pymysql (GH24988).
- Fixed regression in `Index.intersection` incorrectly sorting the values by default (GH24959).
- Fixed regression in `merge()` when merging an empty `DataFrame` with multiple timezone-aware columns on one of the timezone-aware columns (GH25014).
- Fixed regression in `Series.rename_axis()` and `DataFrame.rename_axis()` where passing `None` failed to remove the axis name (GH25034)
- Fixed regression in `to_datetime()` with `box=False` incorrectly returning a `datetime64` object instead of a `timedelta64` object (GH24961)
- Fixed regression where custom hashable types could not be used as column keys in `DataFrame.set_index()` (GH24969)
Bug fixes

Reshaping

- Bug in `DataFrame.groupby()` with `Grouper` when there is a time change (DST) and grouping frequency is '1d' (GH24972)

Visualization

- Fixed the warning for implicitly registered matplotlib converters not showing. See `Restore Matplotlib datetime converter registration` for more (GH24963).

Other

- Fixed AttributeError when printing a DataFrame’s HTML repr after accessing the IPython config object (GH25036)

Contributors

A total of 7 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Alex Buchkovsky
- Roman Yurchak
- h-vetinari
- jbrockmendel
- Jeremy Schendel
- Joris Van den Bossche
- Tom Augspurger

5.7.3 What’s new in 0.24.0 (January 25, 2019)

**Warning:** The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See Dropping Python 2.7 for more details.

This is a major release from 0.23.4 and includes a number of API changes, new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- Optional Integer NA Support
- New APIs for accessing the array backing a Series or Index
- A new top-level method for creating arrays
- Store Interval and Period data in a Series or DataFrame
- Support for joining on two MultiIndexes

Check the API Changes and deprecations before updating.

These are the changes in pandas 0.24.0. See Release notes for a full changelog including other versions of pandas.
Enhancements

Optional integer NA support

pandas has gained the ability to hold integer dtypes with missing values. This long requested feature is enabled through the use of *extension types*.

**Note:** IntegerArray is currently experimental. Its API or implementation may change without warning.

We can construct a Series with the specified dtype. The dtype string Int64 is a pandas ExtensionDtype. Specifying a list or array using the traditional missing value marker of np.nan will infer to integer dtype. The display of the Series will also use the NaN to indicate missing values in string outputs. (GH20700, GH20747, GH22441, GH21789, GH22346)

```
In [1]: s = pd.Series([1, 2, np.nan], dtype='Int64')

In [2]: s
Out[2]:
0    1
1    2
2   <NA>
Length: 3, dtype: Int64
```

Operations on these dtypes will propagate NaN as other pandas operations.

```
# arithmetic
In [3]: s + 1
Out[3]:
0    2
1    3
2   <NA>
Length: 3, dtype: Int64

# comparison
In [4]: s == 1
Out[4]:
0   True
1  False
2   <NA>
Length: 3, dtype: boolean

# indexing
In [5]: s.iloc[1:3]
Out[5]:
1    2
2   <NA>
Length: 2, dtype: Int64

# operate with other dtypes
In [6]: s + s.iloc[1:3].astype('Int8')
Out[6]:
0   <NA>
(continues on next page)```
In [7]: s + 0.01
Out[7]:
0  1.01
1  2.01
2  <NA>
Length: 3, dtype: Float64

These dtypes can operate as part of a DataFrame.

In [8]: df = pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab'))

In [9]: df
Out[9]:
   A  B  C
0  1  1  a
1  2  1  a
2  <NA>  3  b
[3 rows x 3 columns]

In [10]: df.dtypes
Out[10]:
A   Int64
B   int64
C   object
Length: 3, dtype: object

These dtypes can be merged, reshaped, and casted.

In [11]: pd.concat([df[['A']], df[['B', 'C']]], axis=1).dtypes
Out[11]:
A   Int64
B   int64
C   object
Length: 3, dtype: object

In [12]: df['A'].astype(float)
Out[12]:
   0   1.0
   1   2.0
   2   NaN
Name: A, Length: 3, dtype: float64

Reduction and groupby operations such as sum work.

In [13]: df.sum()
Out[13]:
A   3
B 5
C aab
Length: 3, dtype: object

In [14]: df.groupby('B').A.sum()
Out[14]:
   B
1  3
3  0
Name: A, Length: 2, dtype: Int64

**Warning:** The Integer NA support currently uses the capitalized dtype version, e.g. `Int8` as compared to the traditional `int8`. This may be changed at a future date.

See *Nullable integer data type* for more.

### Accessing the values in a Series or Index

`Series.array` and `Index.array` have been added for extracting the array backing a *Series* or *Index*. *(GH19954, GH23623)*

In [15]: idx = pd.period_range('2000', periods=4)
In [16]: idx.array
Out[16]:
<PeriodArray>
Length: 4, dtype: period[D]

In [17]: pd.Series(idx).array
Out[17]:
<PeriodArray>
Length: 4, dtype: period[D]

Historically, this would have been done with `series.values`, but with `.values` it was unclear whether the returned value would be the actual array, some transformation of it, or one of pandas custom arrays (like `Categorical`). For example, with `PeriodIndex`, `.values` generates a new `ndarray` of period objects each time.

In [18]: idx.values
Out[18]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
       Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)

In [19]: id(idx.values)
Out[19]: 139832573479216

In [20]: id(idx.values)
Out[20]: 139833641453104
If you need an actual NumPy array, use `Series.to_numpy()` or `Index.to_numpy()`.

```python
In [21]: idx.to_numpy()
Out[21]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
         Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)
```

```python
In [22]: pd.Series(idx).to_numpy()
Out[22]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
         Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)
```

For Series and Indexes backed by normal NumPy arrays, `Series.array` will return a new `arrays.PandasArray`, which is a thin (no-copy) wrapper around a `numpy.ndarray`. `PandasArray` isn’t especially useful on its own, but it does provide the same interface as any extension array defined in pandas or by a third-party library.

```python
In [23]: ser = pd.Series([1, 2, 3])
In [24]: ser.array
Out[24]: <PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
In [25]: ser.to_numpy()
Out[25]: array([1, 2, 3])
```

We haven’t removed or deprecated `Series.values` or `DataFrame.values`, but we highly recommend and using `.array` or `.to_numpy()` instead.

See `Dtypes` and `Attributes and Underlying Data` for more.

**pandas.array: a new top-level method for creating arrays**

A new top-level method `array()` has been added for creating 1-dimensional arrays (GH22860). This can be used to create any extension array, including extension arrays registered by 3rd party libraries. See the `dtypes docs` for more on extension arrays.

```python
In [26]: pd.array([1, 2, np.nan], dtype='Int64')
Out[26]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

```python
In [27]: pd.array(['a', 'b', 'c'], dtype='category')
Out[27]:
['a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Passing data for which there isn’t dedicated extension type (e.g. float, integer, etc.) will return a new `arrays.PandasArray`, which is just a thin (no-copy) wrapper around a `numpy.ndarray` that satisfies the pandas extension array interface.
On their own, a PandasArray isn’t a very useful object. But if you need write low-level code that works generically for any ExtensionArray, PandasArray satisfies that need.

Notice that by default, if no dtype is specified, the dtype of the returned array is inferred from the data. In particular, note that the first example of [1, 2, np.nan] would have returned a floating-point array, since NaN is a float.

Storing Interval and Period data in Series and DataFrame

Interval and Period data may now be stored in a Series or DataFrame, in addition to an IntervalIndex and PeriodIndex like previously (GH19453, GH22862).

For periods:

For periods:
Previously, these would be cast to a NumPy array with object dtype. In general, this should result in better performance when storing an array of intervals or periods in a Series or column of a DataFrame.

Use Series.array to extract the underlying array of intervals or periods from the Series:

```
In [36]: ser.array
Out[36]:
<IntervalArray>
[(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]
Length: 5, dtype: interval[int64, right]

In [37]: pser.array
Out[37]:
<PeriodArray>
Length: 5, dtype: period[D]
```

These return an instance of arrays.IntervalArray or arrays.PeriodArray, the new extension arrays that back interval and period data.

**Warning:** For backwards compatibility, Series.values continues to return a NumPy array of objects for Interval and Period data. We recommend using Series.array when you need the array of data stored in the Series, and Series.to_numpy() when you know you need a NumPy array.

See Dtypes and Attributes and Underlying Data for more.

### Joining with two multi-indexes

DataFrame.merge() and DataFrame.join() can now be used to join multi-indexed DataFrame instances on the overlapping index levels (GH6360)

See the Merge, join, and concatenate documentation section.

```
In [38]: index_left = pd.MultiIndex.from_tuples([(K0, 'X0'), (K0, 'X1'),
                                                (K1, 'X2')],
                                               names=['key', 'X'])

In [39]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                        'B': ['B0', 'B1', 'B2']}, index=index_left)

In [40]: index_right = pd.MultiIndex.from_tuples([(K0, 'Y0'), (K1, 'Y1'),
                                                (K2, 'Y2'), (K2, 'Y3')],
                                               names=['key', 'Y'])

In [41]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                           'D': ['D0', 'D1', 'D2', 'D3']}, index=index_right)

In [42]: left.join(right)
```

(continues on next page)
For earlier versions this can be done using the following.

```python
In [43]: pd.merge(left.reset_index(), right.reset_index(),
....:           on=['key'], how='inner').set_index(['key', 'X', 'Y'])
Out[43]:
   A  B  C  D
key X  Y
K0 X0 Y0 A0 B0 C0 D0
   X1 Y0 A1 B1 C0 D0
K1 X2 Y1 A2 B2 C1 D1
[3 rows x 4 columns]
```

**Function `read_html` enhancements**

`read_html()` previously ignored `colspan` and `rowspan` attributes. Now it understands them, treating them as sequences of cells with the same value. (GH17054)

```python
In [44]: result = pd.read_html(""
....:    <table>
....:    <thead>
....:      <tr>
....:        <th>A</th><th>B</th><th>C</th>
....:      </tr>
....:    </thead>
....:    <tbody>
....:      <tr>
....:        <td colspan="2">1</td><td>2</td>
....:      </tr>
....:    </tbody>
....:    ""

Previous behavior:

```python
In [13]: result
Out [13]:
[ A  B  C
  0  1  2 NaN]
```

New behavior:
New **Styler.pipe()** method

The `Styler` class has gained a `pipe()` method. This provides a convenient way to apply users’ predefined styling functions, and can help reduce “boilerplate” when using DataFrame styling functionality repeatedly within a notebook. (GH23229)

```python
In [46]: df = pd.DataFrame({'N': [1250, 1500, 1750], 'X': [0.25, 0.35, 0.50]})

In [47]: def format_and_align(styler):
   ....:     return (styler.format({'N': ',', 'X': '{:.1%}'}))
   ....:     .set_properties(**{'text-align': 'right'})
   ....:

In [48]: df.style.pipe(format_and_align).set_caption('Summary of results.

Out[48]: <pandas.io.formats.style.Styler at 0x7f2d17488ee0>
```

Similar methods already exist for other classes in pandas, including `DataFrame.pipe()`, `GroupBy.pipe()`, and `Resampler.pipe()`.

**Renaming names in a MultiIndex**

`DataFrame.rename_axis()` now supports `index` and `columns` arguments and `Series.rename_axis()` supports `index` argument (GH19978). This change allows a dictionary to be passed so that some of the names of a MultiIndex can be changed.

Example:

```python
In [49]: mi = pd.MultiIndex.from_product([list('AB'), list('CD'), list('EF')],
   ....:     names=['AB', 'CD', 'EF'])

In [50]: df = pd.DataFrame(list(range(len(mi))), index=mi, columns=['N'])

In [51]: df

Out[51]:
```

(continues on next page)
D  E  6  
   F  7  

[8 rows x 1 columns]

In [52]: df.rename_axis(index={'CD': 'New'})

Out[52]:

   New
A  C  E  0
   F  1
B  C  E  4
   F  5
D  E  6
   F  7

[8 rows x 1 columns]

See the Advanced documentation on renaming for more details.

Other enhancements

- `merge()` now directly allows merge between objects of type DataFrame and named Series, without the need to convert the Series object into a DataFrame beforehand (GH21220)
- ExcelWriter now accepts mode as a keyword argument, enabling append to existing workbooks when using the openpyxl engine (GH3441)
- FrozenList has gained the .union() and .difference() methods. This functionality greatly simplifies groupby’s that rely on explicitly excluding certain columns. See Splitting an object into groups for more information (GH15475, GH15506).
- DataFrame.to_parquet() now accepts index as an argument, allowing the user to override the engine’s default behavior to include or omit the dataframe’s indexes from the resulting Parquet file. (GH20768)
- read_feather() now accepts columns as an argument, allowing the user to specify which columns should be read. (GH24025)
- DataFrame.corr() and Series.corr() now accept a callable for generic calculation methods of correlation, e.g. histogram intersection (GH22684)
- DataFrame.to_string() now accepts decimal as an argument, allowing the user to specify which decimal separator should be used in the output. (GH23614)
- DataFrame.to_html() now accepts render_links as an argument, allowing the user to generate HTML with links to any URLs that appear in the DataFrame. See the section on writing HTML in the IO docs for example usage. (GH2679)
- pandas.read_csv() now supports pandas extension types as an argument to dtype, allowing the user to use pandas extension types when reading CSVs. (GH23228)
- The shift() method now accepts fill_value as an argument, allowing the user to specify a value which will be used instead of NA/NaT in the empty periods. (GH15486)
- to_datetime() now supports the %Z and %z directive when passed into format (GH13486)
• Series.mode() and DataFrame.mode() now support the dropna parameter which can be used to specify whether NaN/NaT values should be considered (GH17534)

• DataFrame.to_csv() and Series.to_csv() now support the compression keyword when a file handle is passed. (GH21227)

• Index.droplevel() is now implemented also for flat indexes, for compatibility with MultiIndex (GH21115)

• Series.droplevel() and DataFrame.droplevel() are now implemented (GH20342)

• Added support for reading from/writing to Google Cloud Storage via the gcsfs library (GH19454, GH23094)

• DataFrame.to_gbq() and read_gbq() signature and documentation updated to reflect changes from the pandas-gbq library version 0.8.0. Adds a credentials argument, which enables the use of any kind of google-auth credentials. (GH21627, GH22557, GH23662)

• New method HDFStore.walk() will recursively walk the group hierarchy of an HDF5 file (GH10932)

• read_html() copies cell data across colspan and rowspan, and it treats all-th table rows as headers if header kwarg is not given and there is no thead (GH17054)

• Series.nlargest(), Series.nsmallest(), DataFrame.nlargest(), and DataFrame.nsmallest() now accept the value "all" for the keep argument. This keeps all ties for the nth largest/smallest value (GH16818)

• IntervalIndex has gained the set_closed() method to change the existing closed value (GH21670)

• to_csv(), to_csv(), to_json(), and to_json() now support compression='infer' to infer compression based on filename extension (GH15008). The default compression for to_csv, to_json, and to_pickle methods has been updated to 'infer' (GH22004).

• DataFrame.to_sql() now supports writing TIMESTAMP WITH TIME ZONE types for supported databases. For databases that don’t support timezones, datetime data will be stored as timezone unaware local timestamps. See the Datetime data types for implications (GH9086).

• to_timedelta() now supports iso-formatted timedelta strings (GH21877)

• Series and DataFrame now support Iterable objects in the constructor (GH2193)

• DatetimeIndex has gained the DatetimeIndex.timetz attribute. This returns the local time with timezone information. (GH21358)

• round(), ceil(), and floor() for DatetimeIndex and Timestamp now support an ambiguous argument for handling datetimes that are rounded to ambiguous times (GH18946) and a nonexistent argument for handling datetimes that are rounded to nonexistent times. See Nonexistent times when localizing (GH22647)

• The result of resample() is now iterable similar to groupby() (GH15314).

• Series.resample() and DataFrame.resample() have gained the pandas.core.resample.Resampler.quartile() (GH15023).

• DataFrame.resample() and Series.resample() with a PeriodIndex will now respect the base argument in the same fashion as with a DatetimeIndex. (GH23882)

• pandas.api.types.is_list_like() has gained a keyword allow_sets which is True by default; if False, all instances of set will not be considered “list-like” anymore (GH23061)

• Index.to_frame() now supports overriding column name(s) (GH22580).

• Categorical.from_codes() now can take a dtype parameter as an alternative to passing categories and ordered (GH24398).

• New attribute __git_version__ will return git commit sha of current build (GH21295).

• Compatibility with Matplotlib 3.0 (GH22790).
- Added `Interval.overlaps()`, `arrays.IntervalArray.overlaps()`, and `IntervalIndex.overlaps()` for determining overlaps between interval-like objects (GH21998)
- `read_fwf()` now accepts keyword `infer_nrows` (GH15138).
- `to_parquet()` now supports writing a `DataFrame` as a directory of parquet files partitioned by a subset of the columns when `engine = 'pyarrow'` (GH23283)
- `Timestamp.tz_localize()`, `DatetimeIndex.tz_localize()`, and `Series.tz_localize()` have gained the nonexistent argument for alternative handling of nonexistent times. See Nonexistent times when localizing (GH8917, GH24466)
- `Index.difference()`, `Index.intersection()`, `Index.union()`, and `Index.symmetric_difference()` now have an optional sort parameter to control whether the results should be sorted if possible (GH17839, GH24471)
- `read_excel()` now accepts `usecols` as a list of column names or callable (GH18273)
- `MultiIndex.to_flat_index()` has been added to flatten multiple levels into a single-level `Index` object.
- `DataFrame.to_stata()` and `pandas.io.stata.StataWriter117` can write mixed string columns to Stata strl format (GH23633)
- `DataFrame.between_time()` and `DataFrame.at_time()` have gained the `axis` parameter (GH8839)
- `DataFrame.to_records()` now accepts `index_dtypes` and `column_dtypes` parameters to allow different data types in stored column and index records (GH18146)
- `IntervalIndex` has gained the `is_overlapping` attribute to indicate if the `IntervalIndex` contains any overlapping intervals (GH23309)
- `pandas.DataFrame.to_sql()` has gained the `method` argument to control SQL insertion clause. See the insertion method section in the documentation. (GH8953)
- `DataFrame.corrwith()` now supports Spearman’s rank correlation, Kendall’s tau as well as callable correlation methods. (GH21925)
- `DataFrame.to_json()`, `DataFrame.to_csv()`, `DataFrame.to_pickle()`, and other export methods now support tilde(~) in path argument. (GH23473)

**Backwards incompatible API changes**

pandas 0.24.0 includes a number of API breaking changes.

**Increased minimum versions for dependencies**

We have updated our minimum supported versions of dependencies (GH21242, GH18742, GH23774, GH24767). If installed, we now require:
Additionally we no longer depend on feather-format for feather based storage and replaced it with references to pyarrow (GH21639 and GH23053).

**os.linesep is used for line_terminator of DataFrame.to_csv**

`DataFrame.to_csv()` now uses `os.linesep()` rather than '\n' for the default line terminator (GH20353). This change only affects when running on Windows, where '\r\n' was used for line terminator even when '\n' was passed in `line_terminator`.

*Previous behavior* on Windows:

```python
In [1]: data = pd.DataFrame({
   "string_with_lf": ["a
bc"],
   ...
   "string_with_crlf": ["a\r\nbc"]})

In [2]: # When passing file PATH to to_csv,
   ...: # line_terminator does not work, and csv is saved with '\r\n'.
   ...: # Also, this converts all '\n's in the data to '\r\n'.
   ...: data.to_csv("test.csv", index=False, line_terminator='\n')

In [3]: with open("test.csv", mode='rb') as f:
   ...:    print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\n\n"a\r
bc","a\r\nbc"

In [4]: # When passing file OBJECT with newline option to
   ...: # to_csv, line_terminator works.
   ...: with open("test2.csv", mode='w', newline='\n') as f:
   ...:    data.to_csv(f, index=False, line_terminator='\n')

In [5]: with open("test2.csv", mode='rb') as f:
   ...:    print(f.read())
Out[5]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"

New behavior on Windows:

Passing `line_terminator` explicitly, set the line terminator to that character.

```python
In [1]: data = pd.DataFrame({
   "string_with_lf": ["a\nbc"],
   ...
   "string_with_crlf": ["a\r\nbc"]})
```
In [2]: data.to_csv("test.csv", index=False, line_terminator='\n')

In [3]: with open("test.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf
"a\nbc","a\r\nbc"\n'

On Windows, the value of os.linesep is '\r\n', so if line_terminator is not set, '\r\n' is used for line terminator.

In [1]: data = pd.DataFrame({'string_with_lf': ["a\nbc"],
   ...:                      'string_with_crlf': ["a\r\nbc"]})

In [2]: data.to_csv("test.csv", index=False)

In [3]: with open("test.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'

For file objects, specifying newline is not sufficient to set the line terminator. You must pass in the line_terminator explicitly, even in this case.

In [1]: data = pd.DataFrame({'string_with_lf': ["a\nbc"],
   ...:                      'string_with_crlf': ["a\r\nbc"]})

In [2]: with open("test2.csv", mode='w', newline='\n') as f:
   ...:     data.to_csv(f, index=False)

In [3]: with open("test2.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'

---

**Proper handling of np.NaN in a string data-typed column with the Python engine**

There was bug in read_excel() and read_csv() with the Python engine, where missing values turned to 'nan' with dtype=str and na_filter=True. Now, these missing values are converted to the string missing indicator, np.nan. (GH20377)

*Previous behavior:*

In [5]: data = 'a,b,c\n1,,3
4,5,6'
In [6]: df = pd.read_csv(StringIO(data), engine='python', dtype=str, na_filter=True)
In [7]: df.loc[0, 'b']
Out[7]: 'nan'

*New behavior:*

In [53]: data = 'a,b,c\n1,,3\n4,5,6'
In [54]: df = pd.read_csv(StringIO(data), engine='python', dtype=str, na_filter=True)
In [55]: df.loc[0, 'b']
Out[55]: nan

Notice how we now instead output `np.nan` itself instead of a stringified form of it.

### Parsing datetime strings with timezone offsets

Previously, parsing datetime strings with UTC offsets with `to_datetime()` or `DatetimeIndex` would automatically convert the datetime to UTC without timezone localization. This is inconsistent from parsing the same datetime string with `Timestamp` which would preserve the UTC offset in the `tz` attribute. Now, `to_datetime()` preserves the UTC offset in the `tz` attribute when all the datetime strings have the same UTC offset (GH17697, GH11736, GH22457)

**Previous behavior:**

```python
In [2]: pd.to_datetime("2015-11-18 15:30:00+05:30")
Out[2]: Timestamp('2015-11-18 10:00:00')

In [3]: pd.Timestamp("2015-11-18 15:30:00+05:30")
Out[3]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')

# Different UTC offsets would automatically convert the datetimes to UTC (without a UTC timezone)
In [4]: pd.to_datetime(["2015-11-18 15:30:00+05:30", "2015-11-18 16:30:00+06:30"])
Out[4]: DatetimeIndex(['2015-11-18 10:00:00', '2015-11-18 10:00:00'], dtype='datetime64[ns]', freq=None)
```

**New behavior:**

```python
In [56]: pd.to_datetime("2015-11-18 15:30:00+05:30")
Out[56]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')

In [57]: pd.Timestamp("2015-11-18 15:30:00+05:30")
Out[57]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')
```

Parsing datetime strings with the same UTC offset will preserve the UTC offset in the `tz`

```python
In [58]: pd.to_datetime(["2015-11-18 15:30:00+05:30"] * 2)
Out[58]: DatetimeIndex(['2015-11-18 15:30:00+05:30', '2015-11-18 15:30:00+05:30'], dtype='datetime64[ns, pytz.FixedOffset(330)]', freq=None)
```

Parsing datetime strings with different UTC offsets will now create an Index of `datetime.datetime` objects with different UTC offsets

```python
In [59]: idx = pd.to_datetime(["2015-11-18 15:30:00+05:30",
                      ....: "2015-11-18 16:30:00+06:30"])

In [60]: idx[0]
Out[60]: datetime.datetime(2015, 11, 18, 15, 30, tzinfo=tzoffset(None, 19800))
```

(continues on next page)
In [62]: idx[1]
Out[62]: datetime.datetime(2015, 11, 18, 16, 30, tzinfo=tzoffset(None, 23400))

Passing utc=True will mimic the previous behavior but will correctly indicate that the dates have been converted to UTC.

In [63]: pd.to_datetime(["2015-11-18 15:30:00+05:30",
                  ....:
                  "2015-11-18 16:30:00+06:30"], utc=True)
Out[63]: DatetimeIndex(['2015-11-18 10:00:00+00:00', '2015-11-18 10:00:00+00:00'], dtype=dtype('datetime64[ns, UTC]'), freq=None)

Parsing mixed-timezones with read_csv()

read_csv() no longer silently converts mixed-timezone columns to UTC (GH24987).

*Previous behavior*

```python
>>> import io
>>> content = '''
... a
... 2000-01-01T00:00:00+05:00
... 2000-01-01T00:00:00+06:00"
...'''

>>> df = pd.read_csv(io.StringIO(content), parse_dates=['a'])
>>> df.a
0 1999-12-31 19:00:00
1 1999-12-31 18:00:00
Name: a, dtype: datetime64[ns]
```

*New behavior*

```python
In [64]: import io

In [65]: content = '''
... a
... 2000-01-01T00:00:00+05:00
... 2000-01-01T00:00:00+06:00"
...'''

In [66]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'])

In [67]: df.a
Out[67]:
0 2000-01-01 00:00:00+05:00
1 2000-01-01 00:00:00+06:00
Name: a, Length: 2, dtype: object
```

As can be seen, the dtype is object; each value in the column is a string. To convert the strings to an array of datetimes, the date_parser argument
In [68]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'],
    ....:                     date_parser=lambda col: pd.to_datetime(col, utc=True))
....:
In [69]: df.a
Out[69]:
0  1999-12-31 19:00:00+00:00
1  1999-12-31 18:00:00+00:00
Name: a, Length: 2, dtype: datetime64[ns, UTC]

See Parsing datetime strings with timezone offsets for more.

Time values in `dt.end_time` and `to_timestamp(how='end')`

The time values in `Period` and `PeriodIndex` objects are now set to `23:59:59.999999999` when calling `Series.dt.end_time`, `Period.end_time`, `PeriodIndex.end_time`, `Period.to_timestamp()` with `how='end'`, or `PeriodIndex.to_timestamp()` with `how='end'` (GH17157)

Previous behavior:

```
In [2]: p = pd.Period('2017-01-01', 'D')
In [3]: pi = pd.PeriodIndex([p])
In [4]: pd.Series(pi).dt.end_time[0]
Out[4]: Timestamp(2017-01-01 00:00:00)
In [5]: p.end_time
```

New behavior:

Calling `Series.dt.end_time` will now result in a time of `23:59:59.999999999` as is the case with `Period.end_time`, for example

```
In [70]: p = pd.Period('2017-01-01', 'D')
In [71]: pi = pd.PeriodIndex([p])
In [72]: pd.Series(pi).dt.end_time[0]
Out[72]: Timestamp('2017-01-01 23:59:59.999999999')
In [73]: p.end_time
Out[73]: Timestamp('2017-01-01 23:59:59.999999999')
```
Series.unique for timezone-aware data

The return type of `Series.unique()` for datetime with timezone values has changed from an `numpy.ndarray` of `Timestamp` objects to a `arrays.DatetimeArray` (GH24024).

```
In [74]: ser = pd.Series([pd.Timestamp('2000', tz='UTC'),
                .....:               pd.Timestamp('2000', tz='UTC')])
                .....:

Previous behavior:
```

```
In [3]: ser.unique()
Out[3]: array([Timestamp('2000-01-01 00:00:00+0000', tz='UTC')], dtype=object)
```

```
New behavior:
```
```
In [75]: ser.unique()
Out[75]: <DatetimeArray>
[ '2000-01-01 00:00:00+00:00']
Length: 1, dtype: datetime64[ns, UTC]
```

Sparse data structure refactor

SparseArray, the array backing SparseSeries and the columns in a SparseDataFrame, is now an extension array (GH21978, GH19056, GH22835). To conform to this interface and for consistency with the rest of pandas, some API breaking changes were made:

- SparseArray is no longer a subclass of `numpy.ndarray`. To convert a SparseArray to a NumPy array, use `numpy.asarray()`.
- SparseArray.dtype and SparseSeries.dtype are now instances of SparseDtype, rather than np.dtype. Access the underlying dtype with SparseDtype.subtype.
- `numpy.asarray(sparse_array)` now returns a dense array with all the values, not just the non-fill-value values (GH14167)
- SparseArray.take now matches the API of `pandas.api.extensions.ExtensionArray.take()` (GH19506):
  - The default value of allow_fill has changed from False to True.
  - The out and mode parameters are now longer accepted (previously, this raised if they were specified).
  - Passing a scalar for indices is no longer allowed.
- The result of `concat()` with a mix of sparse and dense Series is a Series with sparse values, rather than a SparseSeries.
- SparseDataFrame.combine and DataFrame.combine_first no longer supports combining a sparse column with a dense column while preserving the sparse subtype. The result will be an object-dtype SparseArray.
- Setting SparseArray.fill_value to a fill value with a different dtype is now allowed.
- `DataFrame[column]` is now a Series with sparse values, rather than a SparseSeries, when slicing a single column with sparse values (GH23559).
- The result of `Series.where()` is now a Series with sparse values, like with other extension arrays (GH24077)
Some new warnings are issued for operations that require or are likely to materialize a large dense array:

- A `errors.PerformanceWarning` is issued when using fillna with a method, as a dense array is constructed to create the filled array. Filling with a value is the efficient way to fill a sparse array.

- A `errors.PerformanceWarning` is now issued when concatenating sparse Series with differing fill values. The fill value from the first sparse array continues to be used.

In addition to these API breaking changes, many Performance Improvements and Bug Fixes have been made.

Finally, a Series.sparse accessor was added to provide sparse-specific methods like `Series.sparse.from_coo()`.

```python
In [76]: s = pd.Series([0, 0, 1, 1, 1], dtype='Sparse[int]')
In [77]: s.sparse.density
Out[77]: 0.6
```

### `get_dummies()` always returns a DataFrame

Previously, when `sparse=True` was passed to `get_dummies()`, the return value could be either a `DataFrame` or a `SparseDataFrame`, depending on whether all or a just a subset of the columns were dummy-encoded. Now, a `DataFrame` is always returned (GH24284).

**Previous behavior**

The first `get_dummies()` returns a `DataFrame` because the column A is not dummy encoded. When just ["B", "C"] are passed to `get_dummies`, then all the columns are dummy-encoded, and a `SparseDataFrame` was returned.

```python
In [2]: df = pd.DataFrame({'A': [1, 2], 'B': ['a', 'b'], 'C': ['a', 'a']})
In [3]: type(pd.get_dummies(df, sparse=True))
Out[3]: pandas.core.frame.DataFrame
In [4]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
Out[4]: pandas.core.sparse.frame.SparseDataFrame
```

**New behavior**

Now, the return type is consistently a `DataFrame`.

```python
In [78]: type(pd.get_dummies(df, sparse=True))
Out[78]: pandas.core.frame.DataFrame
In [79]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
Out[79]: pandas.core.frame.DataFrame
```

**Note:** There's no difference in memory usage between a `SparseDataFrame` and a `DataFrame` with sparse values. The memory usage will be the same as in the previous version of pandas.
Raise ValueError in DataFrame.to_dict(orient='index')

Bug in DataFrame.to_dict() raises ValueError when used with orient='index' and a non-unique index instead of losing data (GH22801)

```
In [80]: df = pd.DataFrame({'a': [1, 2], 'b': [0.5, 0.75]}, index=['A', 'A'])

In [81]: df
Out[81]:
   a  b
  A 1 0.50
  A 2 0.75
[2 rows x 2 columns]

In [82]: df.to_dict(orient='index')
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
  Input In [82], in <cell line: 1>()
----> 1 df.to_dict(orient='index')

File /pandas/pandas/core/frame.py:1953, in DataFrame.to_dict(self, orient, into)
     1951     elif orient == "index":
     1952         if not self.index.is_unique:
     1953             raise ValueError("DataFrame index must be unique for orient='index'.")
     1954         return into_c(
     1955         (t[0], dict(zip(self.columns, t[1:])))
     1956     for t in self.itertuples(name=None)
     1957     )
     1959     else:

ValueError: DataFrame index must be unique for orient='index'.
```

Tick DateOffset normalize restrictions

Creating a Tick object (Day, Hour, Minute, Second, Milli, Micro, Nano) with normalize=True is no longer supported. This prevents unexpected behavior where addition could fail to be monotone or associative. (GH21427)

Previous behavior:

```
In [2]: ts = pd.Timestamp('2018-06-11 18:01:14')

In [3]: ts
Out[3]: Timestamp('2018-06-11 18:01:14')

In [4]: tic = pd.offsets.Hour(n=2, normalize=True)
   ...

In [5]: tic
Out[5]: <2 * Hours>

In [6]: ts + tic
```

(continues on next page)
Out[6]: Timestamp('2018-06-11 00:00:00')

In [7]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[7]: False

**New behavior:**

In [83]: ts = pd.Timestamp('2018-06-11 18:01:14')

In [84]: tic = pd.offsets.Hour(n=2)

In [85]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[85]: True

### Period subtraction

Subtraction of a Period from another Period will give a DateOffset, instead of an integer (GH21314)

**Previous behavior:**

In [2]: june = pd.Period('June 2018')

In [3]: april = pd.Period('April 2018')

In [4]: june - april
Out [4]: 2

**New behavior:**

In [86]: june = pd.Period('June 2018')

In [87]: april = pd.Period('April 2018')

In [88]: june - april
Out[88]: <2 * MonthEnds>

Similarly, subtraction of a Period from a PeriodIndex will now return an Index of DateOffset objects instead of an Int64Index

**Previous behavior:**

In [2]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [3]: pi - pi[0]
Out[3]: Int64Index([0, 1, 2], dtype='int64')

**New behavior:**

In [89]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [90]: pi - pi[0]
Out[90]: Index([<0 * MonthEnds>, <MonthEnd>, <2 * MonthEnds>], dtype='object')
Addition/subtraction of NaN from DataFrame

Adding or subtracting NaN from a DataFrame column with timedelta64[ns] dtype will now raise a TypeError instead of returning all-NaN. This is for compatibility with TimedeltaIndex and Series behavior (GH22163).

```python
In [91]: df = pd.DataFrame([pd.Timedelta(days=1)])
In [92]: df
Out[92]:
0 1 days
[1 rows x 1 columns]

Previous behavior:

```python
In [4]: df = pd.DataFrame([pd.Timedelta(days=1)])
In [5]: df - np.nan
Out[5]:
0 NaN
0 NaN
```

New behavior:

```python
In [2]: df - np.nan
...  
TypeError: unsupported operand type(s) for -: 'TimedeltaIndex' and 'float'
```

DataFrame comparison operations broadcasting changes

Previously, the broadcasting behavior of DataFrame comparison operations (==, !=, ...) was inconsistent with the behavior of arithmetic operations (+, -, ...). The behavior of the comparison operations has been changed to match the arithmetic operations in these cases. (GH22880)

The affected cases are:

- Operating against a 2-dimensional np.ndarray with either 1 row or 1 column will now broadcast the same way a np.ndarray would (GH23000).
- A list or tuple with length matching the number of rows in the DataFrame will now raise ValueError instead of operating column-by-column (GH22880).
- A list or tuple with length matching the number of columns in the DataFrame will now operate row-by-row instead of raising ValueError (GH22880).

```python
In [93]: arr = np.arange(6).reshape(3, 2)
In [94]: df = pd.DataFrame(arr)
In [95]: df
Out[95]:
0 1
0 0 1
```

(continues on next page)
1 2 3
2 4 5

[3 rows x 2 columns]

Previous behavior:

In [5]: df == arr[[0], :]
   ...: # comparison previously broadcast where arithmetic would raise
Out[5]:
   0    1
  0  True  True
  1  False False
  2  False False
In [6]: df + arr[[0], :]
   ... Value Error: Unable to coerce to DataFrame, shape must be (3, 2): given (1, 2)
In [7]: df == (1, 2)
   ...: # length matches number of columns;
   ...: # comparison previously raised where arithmetic would broadcast
   ...
Value Error: Invalid broadcasting comparison [(1, 2)] with block values
In [8]: df + (1, 2)
Out[8]:
   0 1
  0 1 3
  1 3 5
  2 5 7
In [9]: df == (1, 2, 3)
   ...: # length matches number of rows
   ...: # comparison previously broadcast where arithmetic would raise
Out[9]:
   0    1
  0  False  True
  1  True  False
  2  False  False
In [10]: df + (1, 2, 3)
   ...
Value Error: Unable to coerce to Series, length must be 2: given 3

New behavior:

# Comparison operations and arithmetic operations both broadcast.
In [96]: df == arr[[0], :]
Out[96]:
   0    1
  0  True  True
  1  False False
  2  False False

(continues on next page)
In [97]: df + arr[[0], :]
Out[97]:
   0  1
   0  0  2
   1  2  4
   2  4  6

In [98]: df == (1, 2)
Out[98]:
   0  1
   0  False False
   1  False False
   2  False False

In [99]: df + (1, 2)
Out[99]:
   0  1
   0  1  3
   1  3  5
   2  5  7

# Comparison operations and arithmetic operations both broadcast.
In [100]: arr = np.arange(6).reshape(3, 2)
In [101]: df = pd.DataFrame(arr)
In [102]: df
Out[102]:
   0  1
   0  0  2
   1  2  4
   2  4  6

# Comparison operations and arithmetic operations both raise ValueError.
In [6]: df == (1, 2, 3)
...:
ValueError: Unable to coerce to Series, length must be 2: given 3
In [7]: df + (1, 2, 3)
...:
ValueError: Unable to coerce to Series, length must be 2: given 3

Dataframe arithmetic operations broadcasting changes

Dataframe arithmetic operations when operating with 2-dimensional np.ndarray objects now broadcast in the same way as np.ndarray broadcast. (GH23000)


**Previous behavior:**

```python
In [5]: df + arr[[0], :]
# 1 row, 2 columns
...
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (1, 2)
In [6]: df + arr[:, [1]]
# 1 column, 3 rows
...
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (3, 1)
```

**New behavior:**

```python
In [103]: df + arr[[0], :]
# 1 row, 2 columns
Out[103]:
   0   1
0  0   2
1  2   4
2  4   6
[3 rows x 2 columns]
In [104]: df + arr[:, [1]]
# 1 column, 3 rows
Out[104]:
   0  1
0  1   2
1  5   6
2  9  10
[3 rows x 2 columns]
```

**Series and Index data-dtype incompatibilities**

Series and Index constructors now raise when the data is incompatible with a passed `dtype`=(GH15832)

**Previous behavior:**

```python
In [4]: pd.Series([-1], dtype="uint64")
Out [4]:
0  18446744073709551615
dtype: uint64
```

**New behavior:**

```python
In [4]: pd.Series([-1], dtype="uint64")
Out [4]:
```

(continues on next page)
OverflowError: Trying to coerce negative values to unsigned integers

## Concatenation changes

Calling `pandas.concat()` on a `Categorical` of ints with NA values now causes them to be processed as objects when concatenating with anything other than another `Categorical` of ints (GH19214)

```python
In [105]: s = pd.Series([0, 1, np.nan])
In [106]: c = pd.Series([0, 1, np.nan], dtype="category")
```

**Previous behavior**

```python
In [3]: pd.concat([s, c])
Out[3]:
0  0.0
1  1.0
2  NaN
0  0.0
1  1.0
2  NaN
dtype: float64
```

**New behavior**

```python
In [107]: pd.concat([s, c])
Out[107]:
0  0.0
1  1.0
2  NaN
0  0.0
1  1.0
2  NaN
Length: 6, dtype: float64
```

## Datetimelike API changes

- For `DatetimeIndex` and `TimedeltaIndex` with non-None `freq` attribute, addition or subtraction of integer-dtyped array or `Index` will return an object of the same class (GH19959)
- `DateOffset` objects are now immutable. Attempting to alter one of these will now raise `AttributeError` (GH21341)
- `PeriodIndex` subtraction of another `PeriodIndex` will now return an object-dtype `Index` of `DateOffset` objects instead of raising a `TypeError` (GH20049)
- `cut()` and `qcut()` now returns a `DatetimeIndex` or `TimedeltaIndex` bins when the input is datetime or timedelta dtype respectively and `retbins=True` (GH19891)
- `DatetimeIndex.to_period()` and `Timestamp.to_period()` will issue a warning when timezone information will be lost (GH21333)
- PeriodIndex.tz_convert() and PeriodIndex.tz_localize() have been removed (GH21781)

Other API changes

- A newly constructed empty DataFrame with integer as the dtype will now only be cast to float64 if index is specified (GH22858)
- Series.str.cat() will now raise if others is a set (GH23009)
- Passing scalar values to DatetimeIndex or TimedeltaIndex will now raise TypeError instead of ValueError (GH23539)
- max_rows and max_cols parameters removed from HTMLFormatter since truncation is handled by DataFrameFormatter (GH23818)
- read_csv() will now raise a ValueError if a column with missing values is declared as having dtype bool (GH20591)
- The column order of the resultant DataFrame from MultiIndex.to_frame() is now guaranteed to match the MultiIndex.names order. (GH22420)
- Incorrectly passing a DatetimeIndex to MultiIndex.from_tuples(), rather than a sequence of tuples, now raises a TypeError rather than a ValueError (GH24024)
- pd.offsets.generate_range() argument time_rule has been removed; use offset instead (GH24157)
- In 0.23.x, pandas would raise a ValueError on a merge of a numeric column (e.g. int dtyped column) and an object dtyped column (GH9780). We have re-enabled the ability to merge object and other dtypes; pandas will still raise on a merge between a numeric and an object dtyped column that is composed only of strings (GH21681)
- Accessing a level of a MultiIndex with a duplicate name (e.g. in get_level_values()) now raises a ValueError instead of a KeyError.
- Invalid construction of IntervalDtype will now always raise a TypeError rather than a ValueError if the subdtype is invalid (GH21185)
- Trying to reindex a DataFrame with a non unique MultiIndex now raises a ValueError instead of an Exception (GH21770)
- Index subtraction will attempt to operate element-wise instead of raising TypeError (GH19369)
- pandas.io.formats.style.Styler supports a number-format property when using to_excel() (GH22015)
- DataFrame.corr() and Series.corr() now raise a ValueError along with a helpful error message instead of a KeyError when supplied with an invalid method (GH22298)
- shift() will now always return a copy, instead of the previous behaviour of returning self when shifting by 0 (GH22397)
- DataFrame.set_index() now gives a better (and less frequent) KeyError, raises a ValueError for incorrect types, and will not fail on duplicate column names with drop=True. (GH22484)
- Slicing a single row of a DataFrame with multiple ExtensionArrays of the same type now preserves the dtype, rather than coercing to object (GH22784)
- DateOffset attribute _cacheable and method _should_cache have been removed (GH23118)
- Series.searchsorted(), when supplied a scalar value to search for, now returns a scalar instead of an array (GH23801).
• Categorical.searchsorted(), when supplied a scalar value to search for, now returns a scalar instead of an array (GH23466).

• Categorical.searchsorted() now raises a KeyError rather than a ValueError, if a searched for key is not found in its categories (GH23466).

• Index.hasnans() and Series.hasnans() now always return a python boolean. Previously, a python or a numpy boolean could be returned, depending on circumstances (GH23294).

• The order of the arguments of DataFrame.to_html() and DataFrame.to_string() is rearranged to be consistent with each other. (GH23614)

• CategoricalIndex.reindex() now raises a ValueError if the target index is non-unique and not equal to the current index. It previously only raised if the target index was not of a categorical dtype (GH23963).

• Series.to_list() and Index.to_list() are now aliases of Series.tolist respectively Index.tolist (GH8826)

• The result of SparseSeries.unstack is now a DataFrame with sparse values, rather than a SparseDataFrame (GH24372).

• DatetimeIndex and TimedeltaIndex no longer ignore the dtype precision. Passing a non-nanosecond resolution dtype will raise a ValueError (GH24753)

Extension type changes

Equality and hashability

pandas now requires that extension dtypes be hashable (i.e. the respective ExtensionDtype objects; hashability is not a requirement for the values of the corresponding ExtensionArray). The base class implements a default __eq__ and __hash__. If you have a parametrized dtype, you should update the ExtensionDtype._metadata tuple to match the signature of your __init__ method. See pandas.api.extensions.ExtensionDtype for more (GH22476).

New and changed methods

• dropna() has been added (GH21185)

• repeat() has been added (GH24349)

• The ExtensionArray constructor, _from_sequence now take the keyword arg copy=False (GH21185)

• pandas.api.extensions.ExtensionArray.shift() added as part of the basic ExtensionArray interface (GH22387).

• searchsorted() has been added (GH24350)

• Support for reduction operations such as sum, mean via opt-in base class method override (GH22762)

• ExtensionArray.isna() is allowed to return an ExtensionArray (GH22325).

Dtype changes

• ExtensionDtype has gained the ability to instantiate from string dtypes, e.g. decimal would instantiate a registered DecimalDtype; furthermore the ExtensionDtype has gained the method construct_array_type (GH21185)

• Added ExtensionDtype._is_numeric for controlling whether an extension dtype is considered numeric (GH22290).

• Added pandas.api.types.register_extension_dtype() to register an extension type with pandas (GH22664)

• Updated the .type attribute for PeriodDtype, DatetimeTZDtype, and IntervalDtype to be instances of the dtype (Period, Timestamp, and Interval respectively) (GH22938)
Operator support

A Series based on an ExtensionArray now supports arithmetic and comparison operators (GH19577). There are two approaches for providing operator support for an ExtensionArray:

1. Define each of the operators on your ExtensionArray subclass.
2. Use an operator implementation from pandas that depends on operators that are already defined on the underlying elements (scalars) of the ExtensionArray.

See the ExtensionArray Operator Support documentation section for details on both ways of adding operator support.

Other changes

- A default repr for pandas.api.extensions.ExtensionArray is now provided (GH23601).
- ExtensionArray._formatting_values() is deprecated. Use ExtensionArray._formatter instead. (GH23601)
- An ExtensionArray with a boolean dtype now works correctly as a boolean indexer. pandas.api.types.is_bool_dtype() now properly considers them boolean (GH2326)

Bug fixes

- Bug in Series.get() for Series using ExtensionArray and integer index (GH21257)
- shift() now dispatches to ExtensionArray.shift() (GH22386)
- Series.combine() works correctly with ExtensionArray inside of Series (GH20825)
- Series.combine() with scalar argument now works for any function type (GH21248)
- Series.astype() and DataFrame.astype() now dispatch to ExtensionArray.astype() (GH21185).
- Slicing a single row of a DataFrame with multiple ExtensionArrays of the same type now preserves the dtype, rather than coercing to object (GH22784)
- Bug when concatenating multiple Series with different extension dtypes not casting to object dtype (GH22994)
- Series backed by an ExtensionArray now work with util.hash_pandas_object() (GH23066)
- DataFrame.stack() no longer converts to object dtype for DataFrames where each column has the same extension dtype. The output Series will have the same dtype as the columns (GH23077).
- Series.unstack() and DataFrame.unstack() no longer convert extension arrays to object-dtype ndarrays. Each column in the output DataFrame will now have the same dtype as the input (GH23077).
- Bug when grouping DataFrame.groupby() and aggregating on ExtensionArray it was not returning the actual ExtensionArray dtype (GH23227).
- Bug in pandas.merge() when merging on an extension array-backed column (GH23020).

Deprecations

- MultiIndex.labels has been deprecated and replaced by MultiIndex.codes. The functionality is unchanged. The new name better reflects the natures of these codes and makes the MultiIndex API more similar to the API for CategoricalIndex (GH13443). As a consequence, other uses of the name labels in MultiIndex have also been deprecated and replaced with codes:
  - You should initialize a MultiIndex instance using a parameter named codes rather than labels.
  - MultiIndex.set_labels has been deprecated in favor of MultiIndex.set_codes().
  - For method MultiIndex.copy(), the labels parameter has been deprecated and replaced by a codes parameter.
- `DataFrame.to_stata()`, `read_stata()`. StataReader and StataWriter have deprecated the encoding argument. The encoding of a Stata dta file is determined by the file type and cannot be changed (GH21244)
- `MultiIndex.to_hierarchical()` is deprecated and will be removed in a future version (GH21613)
- `Series.ptp()` is deprecated. Use `numpy.ptp` instead (GH21614)
- `Series.compress()` is deprecated. Use `Series[condition]` instead (GH18262)
- The signature of `Series.to_csv()` has been uniformed to that of `DataFrame.to_csv()`: the name of the first argument is now `path_or_buf`, the order of subsequent arguments has changed, the `header` argument now defaults to `True`. (GH19715)
- `Categorical.from_codes()` has deprecated providing float values for the `codes` argument. (GH21767)
- `pandas.read_table()` is deprecated. Instead, use `read_csv()` passing `sep='\t'` if necessary. This deprecation has been removed in 0.25.0. (GH21948)
- `Series.str.cat()` has deprecated using arbitrary list-likes within list-likes. A list-like container may still contain many `Series`, `Index` or 1-dimensional `np.ndarray`, or alternatively, only scalar values. (GH21950)
- `FrozenNDArray.searchsorted()` has deprecated the `v` parameter in favor of `value` (GH14645)
- `DatetimeIndex.shift()` and `PeriodIndex.shift()` now accept `periods` argument instead of `n` for consistency with `Index.shift()` and `Series.shift()`. Using `n` throws a deprecation warning (GH22458, GH22912)
- The `fastpath` keyword of the different Index constructors is deprecated (GH23110).
- `Timestamp.tz_localize()`, `DatetimeIndex.tz_localize()`, and `Series.tz_localize()` have deprecated the `errors` argument in favor of the `nonexistent` argument (GH8917)
- The class `FrozenNDArray` has been deprecated. When unpickling, `FrozenNDArray` will be unpickled to `np.ndarray` once this class is removed (GH9031)
- The methods `DataFrame.update()` and `Panel.update()` have deprecated the `raise_conflict=False|True` keyword in favor of `errors='ignore'|'raise'` (GH23585)
- The methods `Series.str.partition()` and `Series.str.rpartition()` have deprecated the `pat` keyword in favor of `sep` (GH22676)
- Deprecated the `nthreads` keyword of `pandas.read_feather()` in favor of `use_threads` to reflect the changes in `pyarrow>=0.11.0` (GH23053)
- `pandas.read_excel()` has deprecated accepting `usecols` as an integer. Please pass in a list of ints from 0 to `usecols` inclusive instead (GH23527)
- Constructing a `TimedeltaIndex` from data with `datetime64`-dtyped data is deprecated, will raise `TypeError` in a future version (GH23539)
- Constructing a `DatetimeIndex` from data with `timedelta64`-dtyped data is deprecated, will raise `TypeError` in a future version (GH23675)
- The `keep_tz=False` option (the default) of the `keep_tz` keyword of `DatetimeIndex.to_series()` is deprecated (GH17832).
- Timezone converting a tz-aware `datetime.datetime` or `Timestamp` with `Timestamp` and the `tz` argument is now deprecated. Instead, use `Timestamp.tz_convert()` (GH23579)
- `pandas.api.types.is_period()` is deprecated in favor of `pandas.api.types.is_period_dtype` (GH23917)
- `pandas.api.types.is_datetime64tz()` is deprecated in favor of `pandas.api.types.is_datetime64tz` (GH23917)
• Creating a TimedeltaIndex, DatetimeIndex, or PeriodIndex by passing range arguments start, end, and periods is deprecated in favor of timedelta_range(), date_range(), or period_range() (GH23919)

• Passing a string alias like 'datetime64[ns, UTC]' as the unit parameter to DatetimeTZDtype is deprecated. Use DatetimeTZDtype.construct_from_string instead (GH23990).

• The skipna parameter of infer_dtype() will switch to True by default in a future version of pandas (GH17066, GH24050)

• In Series.where() with Categorical data, providing an other that is not present in the categories is deprecated. Convert the categorical to a different dtype or add the other to the categories first (GH24077).

• Series.clip_lower(), Series.clip_upper(), DataFrame.clip_lower() and DataFrame.clip_upper() are deprecated and will be removed in a future version. Use Series.clip(lower=threshold), Series.clip(upper=threshold) and the equivalent DataFrame methods (GH24203)

• Series.nonzero() is deprecated and will be removed in a future version (GH18262)

• Passing an integer to Series.fillna() and DataFrame.fillna() with timedelta64[ns] dtypes is deprecated, will raise TypeError in a future version. Use obj.fillna(pd.Timedelta(...)) instead (GH24694)

• Series.cat.categorical, Series.cat.name and Series.cat.index have been deprecated. Use the attributes on Series.cat or Series directly. (GH24751).

• Passing a dtype without a precision like np.dtype('datetime64') or timedelta64 to Index, DatetimeIndex and TimedeltaIndex is now deprecated. Use the nanosecond-precision dtype instead (GH24753).

**Integer addition/subtraction with datetimes and timedeltas is deprecated**

In the past, users could—in some cases—add or subtract integers or integer-dtype arrays from Timestamp, DatetimeIndex and TimedeltaIndex.

This usage is now deprecated. Instead add or subtract integer multiples of the object’s freq attribute (GH21939, GH23878).

**Previous behavior:**

```
In [6]: ts + 2

In [7]: tdi = pd.timedelta_range('1D', periods=2)
In [8]: tdi - np.array([2, 1])
Out[8]: TimedeltaIndex([-1 days, 1 days], dtype='timedelta64[ns]', freq=None)

In [9]: dti = pd.date_range('2001-01-01', periods=2, freq='7D')
In [10]: dti + pd.Index([1, 2])
Out[10]: DatetimeIndex(['2001-01-08', '2001-01-22'], dtype='datetime64[ns]', freq=None)
```

**New behavior:**

```
In [109]: ts + 2 * ts.freq
```
Passing integer data and a timezone to DatetimeIndex

The behavior of `DatetimeIndex` when passed integer data and a timezone is changing in a future version of pandas. Previously, these were interpreted as wall times in the desired timezone. In the future, these will be interpreted as wall times in UTC, which are then converted to the desired timezone (GH24559).

The default behavior remains the same, but issues a warning:

```
In [3]: pd.DatetimeIndex([946684800000000000], tz="US/Central")
/bin/ipython:1: FutureWarning: Passing integer-dtype data and a timezone to DatetimeIndex. Integer values will be interpreted differently in a future version of pandas. Previously, these were viewed as datetime64[ns] values representing the wall time *in the specified timezone*. In the future, these will be viewed as datetime64[ns] values representing the wall time *in UTC*. This is similar to a nanosecond-precision UNIX epoch. To accept the future behavior, use

   pd.to_datetime(integer_data, utc=True).tz_convert(tz)

To keep the previous behavior, use

   pd.to_datetime(integer_data).tz_localize(tz)

#!/bin/python3
Out[3]: DatetimeIndex(['2000-01-01 00:00:00-06:00'], dtype='datetime64[ns, US/Central]', freq=None)
```

As the warning message explains, opt in to the future behavior by specifying that the integer values are UTC, and then converting to the final timezone:

```
In [114]: pd.to_datetime([946684800000000000], utc=True).tz_convert('US/Central')
Out[114]: DatetimeIndex(['1999-12-31 18:00:00-06:00'], dtype='datetime64[ns, US/Central]', freq=None)
```

The old behavior can be retained with by localizing directly to the final timezone:

```
In [115]: pd.to_datetime([946684800000000000]).tz_localize('US/Central')
Out[115]: DatetimeIndex(['2000-01-01 00:00:00-06:00'], dtype='datetime64[ns, US/Central]', freq=None)
```
Converting timezone-aware Series and Index to NumPy arrays

The conversion from a `Series` or `Index` with timezone-aware datetime data will change to preserve timezones by default (GH23569).

NumPy doesn’t have a dedicated dtype for timezone-aware datetimes. In the past, converting a `Series` or `DatetimeIndex` with timezone-aware datetimes would convert to a NumPy array by

1. converting the tz-aware data to UTC
2. dropping the timezone-info
3. returning a `numpy.ndarray` with `datetime64[ns]` dtype

Future versions of pandas will preserve the timezone information by returning an object-dtype NumPy array where each value is a `pandas.Timestamp` with the correct timezone attached

```
In [116]: ser = pd.Series(pd.date_range('2000', periods=2, tz='CET'))

In [117]: ser
Out[117]:
0  2000-01-01 00:00:00+01:00
1  2000-01-02 00:00:00+01:00
Length: 2, dtype: datetime64[ns, CET]
```

The default behavior remains the same, but issues a warning

```
In [8]: np.asarray(ser)
/bin/ipython:1: FutureWarning: Converting timezone-aware DatetimeArray to timezone-naive ndarray with 'datetime64[ns]' dtype. In the future, this will return an ndarray with 'object' dtype where each element is a 'pandas.Timestamp' with the correct 'tz ...

To accept the future behavior, pass 'dtype=object'.
To keep the old behavior, pass 'dtype=datatime64[ns]'.

#!/bin/python3
Out[8]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
dtype='datetime64[ns]')
```

The previous or future behavior can be obtained, without any warnings, by specifying the `dtype`

**Previous behavior**

```
In [118]: np.asarray(ser, dtype='datetime64[ns]')
Out[118]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
dtype='datetime64[ns]')
```

**Future behavior**

```
# New behavior
In [119]: np.asarray(ser, dtype='object')
Out[119]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
     Timestamp('2000-01-02 00:00:00+0100', tz='CET')], dtype=object)
```
Or by using `Series.to_numpy()`

```python
In [120]: ser.to_numpy()
Out[120]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET')], dtype=object)
```

```python
In [121]: ser.to_numpy(dtype="datetime64[ns]"
Out[121]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
       dtype='datetime64[ns]')
```

All the above applies to a `DatetimeIndex` with tz-aware values as well.

**Removal of prior version deprecations/changes**

- The `LongPanel` and `WidePanel` classes have been removed (GH10892)
- `Series.repeat()` has renamed the `reps` argument to `repeats` (GH14645)
- Several private functions were removed from the (non-public) module `pandas.core.common` (GH22001)
- Removal of the previously deprecated module `pandas.core.datetools` (GH14105, GH14094)
- Strings passed into `DataFrame.groupby()` that refer to both column and index levels will raise a `ValueError` (GH14432)
- `Index.repeat()` and `MultiIndex.repeat()` have renamed the `n` argument to `repeats` (GH14645)
- The Series constructor and `.astype` method will now raise a `ValueError` if timestamp dtypes are passed in without a unit (e.g. `np.datetime64`) for the `dtype` parameter (GH15987)
- Removal of the previously deprecated `as_indexer` keyword completely from `str.match()` (GH22356, GH6581)
- The modules `pandas.types`, `pandas.computation`, and `pandas.util.decorators` have been removed (GH16157, GH16250)
- Removed the `pandas.formats.style` shim for `pandas.io.format.style.Styler` (GH16059)
- `pandas.pnow`, `pandas.match`, `pandas.groupby`, `pd.get_store`, `pd.Expr`, and `pd.Term` have been removed (GH15538, GH15940)
- `Categorical.searchsorted()` and `Series.searchsorted()` have renamed the `v` argument to `value` (GH14645)
- `pandas.parser`, `pandas.lib`, and `pandas.tslib` have been removed (GH15537)
- `Index.searchsorted()` have renamed the `key` argument to `value` (GH14645)
- `DataFrame.consolidate` and `Series.consolidate` have been removed (GH15501)
- Removal of the previously deprecated module `pandas.json` (GH19944)
- The module `pandas.tools` has been removed (GH15358, GH16005)
- `SparseArray.get_values()` and `SparseArray.to_dense()` have dropped the `fill` parameter (GH14686)
- `DataFrame.sortlevel` and `Series.sortlevel` have been removed (GH15099)
- `SparseSeries.to_dense()` has dropped the `sparse_only` parameter (GH14686)
- DataFrame.astype() and Series.astype() have renamed the raise_on_error argument to errors (GH14967)
- is_sequence, is_any_int_dtype, and is_floating_dtype have been removed from pandas.api.types (GH16163, GH16189)

Performance improvements

- Slicing Series and DataFrames with an monotonically increasing CategoricalIndex is now very fast and has speed comparable to slicing with an Int64Index. The speed increase is both when indexing by label (using .loc) and position(.iloc) (GH20395) Slicing a monotonically increasing CategoricalIndex itself (i.e. ci[1000: 2000]) shows similar speed improvements as above (GH21659)
- Improved performance of CategoricalIndex.equals() when comparing to another CategoricalIndex (GH24023)
- Improved performance of Series.describe() in case of numeric dtpyes (GH21274)
- Improved performance of pandas.core.groupby.GroupBy.rank() when dealing with tied rankings (GH21237)
- Improved performance of DataFrame.set_index() with columns consisting of Period objects (GH21582, GH21606)
- Improved performance of Series.at() and Index.get_value() for Extension Arrays values (e.g. Categorical) (GH24204)
- Improved performance of membership checks in Categorical and CategoricalIndex (i.e. x in cat-style checks are much faster). CategoricalIndex.contains() is likewise much faster (GH21369, GH21508)
- Improved performance of HDFStore.groups() (and dependent functions like HDFStore.keys(). (i.e. x in store checks are much faster) (GH21372)
- Improved the performance of pandas.get_dummies() with sparse=True (GH21997)
- Improved performance of IndexEngine.get_indexer_non_unique() for sorted, non-unique indexes (GH9466)
- Improved performance of PeriodEngine.get_indexer_non_unique() (GH23083)
- Improved performance of concat() for Series objects (GH23404)
- Improved performance of DatetimeIndex.normalize() and Timestamp.normalize() for timezone naive or UTC datetimes (GH23634)
- Improved performance of DatetimeIndex.tz_localize() and various DatetimeIndex attributes with dateutil UTC timezone (GH23772)
- Fixed a performance regression on Windows with Python 3.7 of read_csv() (GH23516)
- Improved performance of Categorical constructor for Series objects (GH23814)
- Improved performance of where() for Categorical data (GH24077)
- Improved performance of iterating over a Series. Using DataFrame.itertuples() now creates iterators without internally allocating lists of all elements (GH20783)
- Improved performance of Period constructor, additionally benefitting PeriodArray and PeriodIndex creation (GH24084, GH24118)
- Improved performance of tz-aware DatetimeArray binary operations (GH24491)
Bug fixes

Categorical

- Bug in `Categorical.from_codes()` where NaN values in codes were silently converted to 0 (GH21767). In the future this will raise a `ValueError`. Also changes the behavior of `.from_codes([1.1, 2.0])`.

- Bug in `Categorical.sort_values()` where NaN values were always positioned in front regardless of `na_position` value. (GH22556).

- Bug when indexing with a boolean-valued `Categorical`. Now a boolean-valued `Categorical` is treated as a boolean mask (GH22665)

- Constructing a `CategoricalIndex` with empty values and boolean categories was raising a `ValueError` after a change to dtype coercion (GH22702).

- Bug in `Categorical.take()` with a user-provided `fill_value` not encoding the `fill_value`, which could result in a `ValueError`, incorrect results, or a segmentation fault (GH23296).

- In `Series.unstack()`, specifying a `fill_value` not present in the categories now raises a `TypeError` rather than ignoring the `fill_value` (GH23284).

- Bug when resampling `DataFrame.resample()` and aggregating on categorical data, the categorical dtype was getting lost. (GH23227)

- Bug in many methods of the `.str`-accessor, which always failed on calling the `CategoricalIndex.str` constructor (GH23555, GH23556)

- Bug in `Series.where()` losing the categorical dtype for categorical data (GH24077)

- Bug in `Categorical.apply()` where NaN values could be handled unpredictably. They now remain unchanged (GH24241)

- Bug in `Categorical` comparison methods incorrectly raising `ValueError` when operating against a `DataFrame` (GH24630)

- Bug in `Categorical.set_categories()` where setting fewer new categories with `rename=True` caused a segmentation fault (GH24675)

Datetimelike

- Fixed bug where two `DateOffset` objects with different `normalize` attributes could evaluate as equal (GH21404)

- Fixed bug where `Timestamp.resolution()` incorrectly returned 1-microsecond `timedelta` instead of 1-nanosecond `Timedelta` (GH21336, GH21365)

- Bug in `to_datetime()` that did not consistently return an `Index` when `box=True` was specified (GH21864)

- Bug in `DatetimeIndex` comparisons where string comparisons incorrectly raises `TypeError` (GH22074)

- Bug in `DatetimeIndex` comparisons when comparing against `timedelta64[ns]` dtyped arrays; in some cases `TypeError` was incorrectly raised, in others it incorrectly failed to raise (GH22074)

- Bug in `DatetimeIndex` comparisons when comparing against object-dtyped arrays (GH22074)

- Bug in `DataFrame` with `datetime64[ns]` dtype addition and subtraction with `Timedelta`-like objects (GH22005, GH22163)

- Bug in `DataFrame` with `datetime64[ns]` dtype addition and subtraction with `DateOffset` objects returning an `object` dtype instead of `datetime64[ns]` dtype (GH21610, GH22163)
• Bug in DataFrame with datetime64[ns] dtype comparing against NaT incorrectly (GH22242, GH22163)

• Bug in DataFrame with datetime64[ns] dtype subtracting Timestamp-like object incorrectly returned datetime64[ns] dtype instead of timedelta64[ns] dtype (GH8554, GH22163)

• Bug in DataFrame with datetime64[ns] dtype subtracting np.datetime64 object with non-nanosecond unit failing to convert to nanoseconds (GH18874, GH22163)

• Bug in DataFrame comparisons against Timestamp-like objects failing to raise TypeError for inequality checks with mismatched types (GH8932, GH22163)

• Bug in DataFrame with mixed dtypes including datetime64[ns] incorrectly raising TypeError on equality comparisons (GH13128, GH22163)

• Bug in DataFrame.values returning a DatetimeIndex for a single-column DataFrame with tz-aware date-time values. Now a 2-D numpy.ndarray of Timestamp objects is returned (GH24024)

• Bug in DataFrame.eq() comparison against NaT incorrectly returning True or NaN (GH15697, GH22163)

• Bug in DatetimeIndex subtraction that incorrectly failed to raise OverflowError (GH22492, GH22508)

• Bug in DatetimeIndex incorrectly allowing indexing with Timedelta object (GH20464)

• Bug in DatetimeIndex where frequency was being set if original frequency was None (GH22150)

• Bug in rounding methods of DatetimeIndex (round(), ceil(), floor()) and Timestamp (round(), ceil(), floor()) could give rise to loss of precision (GH22591)

• Bug in to_datetime() with an Index argument that would drop the name from the result (GH21697)

• Bug in PeriodIndex where adding or subtracting a timedelta or Tick object produced incorrect results (GH22988)

• Bug in the Series repr with period-dtype data missing a space before the data (GH23601)

• Bug in date_range() when decrementing a start date to a past end date by a negative frequency (GH23270)

• Bug in Series.min() which would return NaN instead of NaT when called on a series of NaT (GH23282)

• Bug in Series.combine_first() not properly aligning categoricals, so that missing values in self where not filled by valid values from other (GH24147)

• Bug in DataFrame.combine() with datetimelike values raising a TypeError (GH23079)

• Bug in date_range() with frequency of Day or higher where dates sufficiently far in the future could wrap around to the past instead of raising OutOfBoundsDatetime (GH14187)

• Bug in period_range() ignoring the frequency of start and end when those are provided as Period objects (GH20535).

• Bug in PeriodIndex with attribute freq.n greater than 1 where adding a DateOffset object would return incorrect results (GH23215)

• Bug in Series that interpreted string indices as lists of characters when setting datetimelike values (GH23451)

• Bug in DataFrame when creating a new column from an ndarray of Timestamp objects with timezones creating an object-dtype column, rather than datetime with timezone (GH23932)

• Bug in Timestamp constructor which would drop the frequency of an input Timestamp (GH22311)

• Bug in DatetimeIndex where calling np.array(dtindex, dtype=object) would incorrectly return an array of long objects (GH23524)

• Bug in Index where passing a timezone-aware DatetimeIndex and dtype=object would incorrectly raise a ValueError (GH23524)
• Bug in `Index` where calling `np.array(dtindex, dtype=object)` on a timezone-naive `DatetimeIndex` would return an array of `datetime` objects instead of `Timestamp` objects, potentially losing nanosecond portions of the timestamps (GH23524)

• Bug in `Categorical.__setitem__` not allowing setting with another `Categorical` when both are unordered and have the same categories, but in a different order (GH24142)

• Bug in `date_range()` where using dates with millisecond resolution or higher could return incorrect values or the wrong number of values in the index (GH24110)

• Bug in `DatetimeIndex` where constructing a `DatetimeIndex` from a `Categorical` or `CategoricalIndex` would incorrectly drop timezone information (GH18664)

• Bug in `DatetimeIndex` and `TimedeltaIndex` where indexing with `Ellipsis` would incorrectly lose the index’s `freq` attribute (GH21282)

• Clarified error message produced when passing an incorrect `freq` argument to `DatetimeIndex` with `NaT` as the first entry in the passed data (GH11587)

• Bug in `to_datetime()` where `box` and `utc` arguments were ignored when passing a `DataFrame` or `dict` of unit mappings (GH23760)

• Bug in `Series.dt` where the cache would not update properly after an in-place operation (GH24408)

• Bug in `PeriodIndex` where comparisons against an array-like object with length 1 failed to raise `ValueError` (GH23078)

• Bug in `DatetimeIndex.astype()`, `PeriodIndex.astype()` and `TimedeltaIndex.astype()` ignoring the sign of the `dtype` for unsigned integer dtypes (GH24405).

• Fixed bug in `Series.max()` with `datetime64[ns]`-dtype failing to return `NaT` when nulls are present and `skipna=False` is passed (GH24265)

• Bug in `to_datetime()` where arrays of `datetime` objects containing both timezone-aware and timezone-naive datetimes would fail to raise `ValueError` (GH24569)

• Bug in `to_datetime()` with invalid datetime format doesn’t coerce input to `NaT` even if `errors='coerce'` (GH24763)

### Timedelta

• Bug in `DataFrame` with `timedelta64[ns]` dtype division by `Timedelta`-like scalar incorrectly returning `timedelta64[ns]` dtype instead of `float64` dtype (GH20088, GH22163)

• Bug in adding a `Index` with object dtype to a `Series` with `timedelta64[ns]` dtype incorrectly raising (GH22390)

• Bug in multiplying a `Series` with numeric dtype against a `timedelta` object (GH22390)

• Bug in `Series` with numeric dtype when adding or subtracting an array or `Series` with `timedelta64` dtype (GH22390)

• Bug in `Index` with numeric dtype when multiplying or dividing an array with dtype `timedelta64` (GH22390)

• Bug in `TimedeltaIndex` incorrectly allowing indexing with `Timestamp` object (GH20464)

• Fixed bug where subtracting `Timedelta` from an object-dtyped array would raise `TypeError` (GH21980)

• Fixed bug in adding a `DataFrame` with all-`timedelta64[ns]` dtypes to a `DataFrame` with all-integer dtypes returning incorrect results instead of raising `TypeError` (GH22696)

• Bug in `TimedeltaIndex` where adding a timezone-aware datetime scalar incorrectly returned a timezone-naive `DatetimeIndex` (GH23215)
• Bug in `TimedeltaIndex` where adding `np.timedelta64('NaT')` incorrectly returned an all-NaT `DatetimeIndex` instead of an all-NaT `TimedeltaIndex` (GH23215)

• Bug in `Timedelta` and `to_timedelta()` have inconsistencies in supported unit string (GH21762)

• Bug in `TimedeltaIndex` division where dividing by another `TimedeltaIndex` raised `TypeError` instead of returning a `Float64Index` (GH23829, GH22631)

• Bug in `TimedeltaIndex` comparison operations where comparing against non-Timedelta-like objects would raise `TypeError` instead of returning all-False for `__eq__` and all-True for `__ne__` (GH24056)

• Bug in `Timedelta` comparisons when comparing with a Tick object incorrectly raising `TypeError` (GH24710)

**Timezones**

• Bug in `Index.shift()` where an `AssertionError` would raise when shifting across DST (GH8616)

• Bug in `Timestamp` constructor where passing an invalid timezone offset designator (Z) would not raise a `ValueError` (GH8910)

• Bug in `Timestamp.replace()` where replacing at a DST boundary would retain an incorrect offset (GH7825)

• Bug in `Series.replace()` with `datetime64[ns, tz]` data when replacing NaT (GH11792)

• Bug in `Timestamp` when passing different string date formats with a timezone offset would produce different timezone offsets (GH12064)

• Bug when comparing a tz-naive `Timestamp` to a tz-aware `DatetimeIndex` which would coerce the `DatetimeIndex` to tz-naive (GH12601)

• Bug in `Series.truncate()` with a tz-aware `DatetimeIndex` which would cause a core dump (GH9243)

• Bug in `Series` constructor which would coerce tz-aware and tz-naive `Timestamp` to tz-aware (GH13051)

• Bug in `Index` with `datetime64[ns, tz]` dtype that did not localize integer data correctly (GH20964)

• Fixed bug where `DataFrame.describe()` and `Series.describe()` on tz-aware datetimes did not show first and last result (GH21328)

• Bug in `DatetimeIndex` comparisons failing to raise `TypeError` when comparing timezone-aware `DatetimeIndex` against `np.datetime64` (GH22074)

• Bug in `DataFrame` assignment with a timezone-aware scalar (GH19843)

• Bug in `DataFrame.asof()` that raised a `TypeError` when attempting to compare tz-naive and tz-aware timestamps (GH21194)

• Bug when constructing a `DatetimeIndex` with `Timestamp` constructed with the replace method across DST (GH18785)

• Bug when setting a new value with `DataFrame.loc()` with a `DatetimeIndex` with a DST transition (GH18308, GH20724)

• Bug in `Index.unique()` that did not re-localize tz-aware dates correctly (GH21737)

• Bug when indexing a `Series` with a DST transition (GH21846)

• Bug in `DataFrame.resample()` and `Series.resample()` where an `AmbiguousTimeError` or `NonExistentTimeError` would raise if a timezone aware timeseries ended on a DST transition (GH19375, GH10117)
• Bug in DataFrame.drop() and Series.drop() when specifying a tz-aware Timestamp key to drop from a DatetimeIndex with a DST transition (GH21761)

• Bug in DatetimeIndex constructor where NaT and dateutil.tz.tzlocal would raise an OutOfBoundsDateTime error (GH23807)

• Bug in DatetimeIndex tz_localize() and Timestamp tz_localize() with dateutil.tz.tzlocal near a DST transition that would return an incorrectly localized datetime (GH23807)

• Bug in Timestamp constructor where a dateutil.tz.tzutc timezone passed with a datetime.datetime argument would be converted to a pytz.UTC timezone (GH23807)

• Bug in to_datetime() where utc=True was not respected when specifying a unit and errors='ignore'(GH23758)

• Bug in to_datetime() where utc=True was not respected when passing a Timestamp (GH24415)

• Bug in DataFrame.any() returns wrong value when axis=1 and the data is of datetimelike type (GH23070)

• Bug in DatetimeIndex.to_period() where a timezone aware index was converted to UTC first before creating PeriodIndex (GH22905)

• Bug in DataFrame tz_localize(), DataFrame tz_convert(), Series tz_localize(), and Series tz_convert() where copy=False would mutate the original argument inplace (GH6326)

• Bug in DataFrame max() and DataFrame min() with axis=1 where a Series with NaN would be returned when all columns contained the same timezone (GH10390)

Offsets

• Bug in FY5253 where date offsets could incorrectly raise an AssertionError in arithmetic operations (GH14774)

• Bug in DateOffset where keyword arguments week and milliseconds were accepted and ignored. Passing these will now raise ValueError (GH19398)

• Bug in adding DateOffset with DataFrame or PeriodIndex incorrectly raising TypeError (GH23215)

• Bug in comparing DateOffset objects with non-DateOffset objects, particularly strings, raising ValueError instead of returning False for equality checks and True for not-equal checks (GH23524)

Numeric

• Bug in Series __rmatmul__ doesn’t support matrix vector multiplication (GH21530)

• Bug in factorize() fails with read-only array (GH12813)

• Fixed bug in unique() handled signed zeros inconsistently: for some inputs 0.0 and -0.0 were treated as equal and for some inputs as different. Now they are treated as equal for all inputs (GH21866)

• Bug in DataFrame agg(), DataFrame transform() and DataFrame apply() where, when supplied with a list of functions and axis=1 (e.g. df.apply(['sum', 'mean'], axis=1)), a TypeError was wrongly raised. For all three methods such calculation are now done correctly. (GH16679).

• Bug in Series comparison against datetime-like scalars and arrays (GH22074)

• Bug in DataFrame multiplication between boolean dtype and integer returning object dtype instead of integer dtype (GH22047, GH22163)

• Bug in DataFrame apply() where, when supplied with a string argument and additional positional or keyword arguments (e.g. df.apply('sum', min_count=1)), a TypeError was wrongly raised (GH22376)
• Bug in `DataFrame.astype()` to extension dtypes may raise `AttributeError` (GH22578)
• Bug in `DataFrame` with `timedelta64[ns]` dtype arithmetic operations with `ndarray` with integer dtype incorrectly treating the array as `timedelta64[ns]` dtype (GH23114)
• Bug in `Series.rpow()` with object dtype `NaN` for `1 ** NA` instead of `1` (GH22922).
• `Series.agg()` can now handle numpy `NaN`-aware methods like `numpy.nansum()` (GH19629)
• Bug in `Series.rank()` and `DataFrame.rank()` when `pct=True` and more than $2^{24}$ rows are present resulted in percentages greater than `1.0` (GH18271)
• Calls such as `DataFrame.round()` with a non-unique `CategoricalIndex()` now return expected data. Previously, data would be improperly duplicated (GH21809).
• Added `log10`, `floor` and `ceil` to the list of supported functions in `DataFrame.eval()` (GH24139, GH24353)
• Logical operations `&`, `|`, `^` between `Series` and `Index` will no longer raise `ValueError` (GH22092)
• Checking PEP 3141 numbers in `is_scalar()` function returns `True` (GH22903)
• Reduction methods like `Series.sum()` now accept the default value of `keepdims=False` when called from a NumPy ufunc, rather than raising a `TypeError`. Full support for `keepdims` has not been implemented (GH24356).

**Conversion**

• Bug in `DataFrame.combine_first()` in which column types were unexpectedly converted to float (GH20699)
• Bug in `DataFrame.clip()` in which column types are not preserved and casted to float (GH24162)
• Bug in `DataFrame.clip()` when order of columns of dataframes doesn’t match, result observed is wrong in numeric values (GH20911)
• Bug in `DataFrame.astype()` where converting to an extension dtype when duplicate column names are present causes a `RecursionError` (GH24704)

**Strings**

• Bug in `Index.str.partition()` was not nan-safe (GH23558).
• Bug in `Index.str.split()` was not nan-safe (GH23677).
• Bug `Series.str.contains()` not respecting the `na` argument for a Categorical dtype `Series` (GH22158)
• Bug in `Index.str.cat()` when the result contained only `NaN` (GH24044)

**Interval**

• Bug in the `IntervalIndex` constructor where the `closed` parameter did not always override the inferred `closed` (GH19370)
• Bug in the `IntervalIndex` repr where a trailing comma was missing after the list of intervals (GH20611)
• Bug in `Interval` where scalar arithmetic operations did not retain the `closed` value (GH22313)
• Bug in `IntervalIndex` where indexing with datetime-like values raised a `KeyError` (GH20636)
• Bug in `IntervalTree` where data containing `NaN` triggered a warning and resulted in incorrect indexing queries with `IntervalIndex` (GH23352)
## Indexing

- Bug in `DataFrame.ne()` fails if columns contain column name “dtype” (GH22383)
- The traceback from a `KeyError` when asking `.loc` for a single missing label is now shorter and more clear (GH21557)
- `PeriodIndex` now emits a `KeyError` when a malformed string is looked up, which is consistent with the behavior of `DatetimeIndex` (GH22803)
- When `.ix` is asked for a missing integer label in a `MultiIndex` with a first level of integer type, it now raises a `KeyError`, consistently with the case of a flat `Int64Index`, rather than falling back to positional indexing (GH21593)
- Bug in `Index.reindex()` when reindexing a tz-naive and tz-aware `DatetimeIndex` (GH8306)
- Bug in `Series.reindex()` when reindexing an empty series with a `datetime64[ns, tz]` dtype (GH20869)
- Bug in `DataFrame` when setting values with `.loc` and a timezone aware `DatetimeIndex` (GH11365)
- `DataFrame.__getitem__` now accepts dictionaries and dictionary keys as list-likes of labels, consistently with `Series.__getitem__` (GH21294)
- Fixed `DataFrame[np.nan]` when columns are non-unique (GH21428)
- Bug when indexing `DatetimeIndex` with nanosecond resolution dates and timezones (GH11679)
- Bug where indexing with a Numpy array containing negative values would mutate the indexer (GH21867)
- Bug where mixed indexes wouldn’t allow integers for `.at` (GH19860)
- `Float64Index.get_loc` now raises `KeyError` when boolean key passed. (GH19087)
- Bug in `DataFrame.loc()` when indexing with an `IntervalIndex` (GH19977)
- `Index` no longer mangles `None`, `NaN` and `NaT`, i.e. they are treated as three different keys. However, for numeric Index all three are still coerced to a `NaN` (GH22332)
- Bug in `scalar` in `Index` if scalar is a float while the `Index` is of integer dtype (GH22085)
- Bug in `MultiIndex.set_levels()` when levels value is not subscriptable (GH23273)
- Bug where setting a timedelta column by `Index` causes it to be casted to double, and therefore lose precision (GH23511)
- Bug in `Index.union()` and `Index.intersection()` where name of the `Index` of the result was not computed correctly for certain cases (GH9943, GH9862)
- Bug in `Index` slicing with boolean `Index` may raise `TypeError` (GH22533)
- Bug in `PeriodArray.__setitem__` when accepting slice and list-like value (GH23978)
- Bug in `DatetimeIndex, TimedeltaIndex` where indexing with Ellipsis would lose their `freq` attribute (GH21282)
- Bug in `iat` where using it to assign an incompatible value would create a new column (GH23236)
Missing

- Bug in `DataFrame.fillna()` where a `ValueError` would raise when one column contained a `datetime64[ns, tz]` dtype (GH15522)
- Bug in `Series.hasnans()` that could be incorrectly cached and return incorrect answers if null elements are introduced after an initial call (GH19700)
- `Series.isin()` now treats all NaN-floats as equal also for np.object-dtype. This behavior is consistent with the behavior for float64 (GH22119)
- `unique()` no longer mangles NaN-floats and the NaT-object for np.object-dtype, i.e. NaT is no longer coerced to a NaN-value and is treated as a different entity. (GH22295)
- `DataFrame` and `Series` now properly handle numpy masked arrays with hardened masks. Previously, constructing a DataFrame or Series from a masked array with a hard mask would create a pandas object containing the underlying value, rather than the expected NaN. (GH24574)
- Bug in `DataFrame` constructor where `dtype` argument was not honored when handling numpy masked record arrays. (GH24874)

MultiIndex

- Bug in `io.formats.style.Styler.applymap()` where `subset=` with `MultiIndex` slice would reduce to `Series` (GH19861)
- Removed compatibility for `MultiIndex` pickles prior to version 0.8.0; compatibility with `MultiIndex` pickles from version 0.13 forward is maintained (GH21654)
- `MultiIndex.get_loc_level()` (and as a consequence, `.loc` on a `Series` or `DataFrame` with a `MultiIndex` index) will now raise a `KeyError`, rather than returning an empty slice, if asked a label which is present in the levels but is unused (GH22221)
- `MultiIndex` has gained the `MultiIndex.from_frame()`, it allows constructing a `MultiIndex` object from a `DataFrame` (GH22420)
- Fix `TypeError` in Python 3 when creating `MultiIndex` in which some levels have mixed types, e.g. when some labels are tuples (GH15457)

IO

- Bug in `read_csv()` in which a column specified with `CategoricalDtype` of boolean categories was not being correctly coerced from string values to booleans (GH20498)
- Bug in `read_csv()` in which unicode column names were not being properly recognized with Python 2.x (GH13253)
- Bug in `DataFrame.to_sql()` when writing timezone aware data (`datetime64[ns, tz]` dtype) would raise a `TypeError` (GH9086)
- Bug in `DataFrame.to_sql()` where a naive `DatetimeIndex` would be written as `TIMESTAMP WITH TIMEZONE` type in supported databases, e.g. PostgreSQL (GH23510)
- Bug in `read_excel()` when `parse_cols` is specified with an empty dataset (GH9208)
- `read_html()` no longer ignores all-whitespace `<tr>` within `<thead>` when considering the `skiprows` and `header` arguments. Previously, users had to decrease their `header` and `skiprows` values on such tables to work around the issue. (GH21641)
• *read_excel()* will correctly show the deprecation warning for previously deprecated *sheetname* (GH17994)
• *read_csv()* and *read_table()* will throw UnicodeError and not coredump on badly encoded strings (GH22748)
• *read_csv()* will correctly parse timezone-aware datetimes (GH22256)
• Bug in *read_csv()* in which memory management was prematurely optimized for the C engine when the data was being read in chunks (GH23509)
• Bug in *read_csv()* in unnamed columns were being improperly identified when extracting a multi-index (GH23687)
• *read_sas()* will parse numbers in sas7bdat-files that have width less than 8 bytes correctly. (GH21616)
• *read_sas()* will correctly parse sas7bdat files with many columns (GH22628)
• *read_sas()* will correctly parse sas7bdat files with data page types having also bit 7 set (so page type is 128 + 256 = 384) (GH16615)
• Bug in *read_sas()* in which an incorrect error was raised on an invalid file format. (GH24548)
• Bug in *detect_client_encoding()* where potential IOError goes unhandled when importing in a mod_wsgi process due to restricted access to stdout. (GH21552)
• Bug in *DataFrame.to_html()* with *index=False* misses truncation indicators (...) on truncated DataFrame (GH15019, GH22783)
• Bug in *DataFrame.to_html()* with *index=False* when both columns and row index are MultiIndex (GH22579)
• Bug in *DataFrame.to_html()* with *index_names=False* displaying index name (GH22747)
• Bug in *DataFrame.to_html()* with *header=False* not displaying row index names (GH23788)
• Bug in *DataFrame.to_html()* with *sparsify=False* that caused it to raise TypeError (GH22887)
• Bug in *DataFrame.to_string()* that broke column alignment when *index=False* and width of first column's values is greater than the width of first column's header (GH16839, GH13032)
• Bug in *DataFrame.to_string()* that caused representations of *DataFrame* to not take up the whole window (GH22984)
• Bug in *DataFrame.to_csv()* where a single level MultiIndex incorrectly wrote a tuple. Now just the value of the index is written (GH19589).
• HDFStore will raise ValueError when the *format* kwarg is passed to the constructor (GH13291)
• Bug in *HDFStore.append()* when appending a *DataFrame* with an empty string column and *min_itemsize < 8* (GH12242)
• Bug in *read_csv()* in which memory leaks occurred in the C engine when parsing NaN values due to insufficient cleanup on completion or error (GH21353)
• Bug in *read_csv()* in which incorrect error messages were being raised when skipfooter was passed in along with *nrows*, *iterator*, or *chunksize* (GH23711)
• Bug in *read_csv()* in which *MultiIndex* index names were being improperly handled in the cases when they were not provided (GH23484)
• Bug in *read_csv()* in which unnecessary warnings were being raised when the dialect's values conflicted with the default arguments (GH23761)
• Bug in *read_html()* in which the error message was not displaying the valid flavors when an invalid one was provided (GH23549)
• Bug in `read_excel()` in which extraneous header names were extracted, even though none were specified (GH11733)

• Bug in `read_excel()` in which column names were not being properly converted to string sometimes in Python 2.x (GH23874)

• Bug in `read_excel()` in which `index_col=None` was not being respected and parsing index columns anyway (GH18792, GH20480)

• Bug in `read_excel()` in which `usecols` was not being validated for proper column names when passed in as a string (GH20480)

• Bug in `DataFrame.to_dict()` when the resulting dict contains non-Python scalars in the case of numeric data (GH23753)

• `DataFrame.to_string()`, `DataFrame.to_html()`, `DataFrame.to_latex()` will correctly format output when a string is passed as the `float_format` argument (GH21625, GH22270)

• Bug in `read_csv()` that caused it to raise `OverflowError` when trying to use ‘inf’ as `na_value` with integer index column (GH17128)

• Bug in `read_csv()` that caused the C engine on Python 3.6+ on Windows to improperly read CSV filenames with accented or special characters (GH15086)

• Bug in `read_fwf()` in which the compression type of a file was not being properly inferred (GH22199)

• Bug in pandas.io.json.json_normalize() that caused it to raise `TypeError` when two consecutive elements of `record_path` are dicts (GH22706)

• Bug in `DataFrame.to_stata()`, pandas.io.stata.StataWriter and pandas.io.stata.StataWriter117 where a exception would leave a partially written and invalid dta file (GH23573)

• Bug in `DataFrame.to_stata()` and pandas.io.stata.StataWriter117 that produced invalid files when using strLs with non-ASCII characters (GH23573)

• Bug in HDFStore that caused it to raise `ValueError` when reading a Dataframe in Python 3 from fixed format written in Python 2 (GH24510)

• Bug in `DataFrame.to_string()` and more generally in the floating `repr` formatter. Zeros were not trimmed if `inf` was present in a columns while it was the case with NA values. Zeros are now trimmed as in the presence of NA (GH24861).

• Bug in the `repr` when truncating the number of columns and having a wide last column (GH24849).

**Plotting**

• Bug in `DataFrame.plot.scatter()` and `DataFrame.plot.hexbin()` caused x-axis label and ticklabels to disappear when colorbar was on in IPython inline backend (GH10611, GH10678, and GH20455)

• Bug in plotting a Series with datetimes using `matplotlib.axes.Axes.scatter()` (GH22039)

• Bug in `DataFrame.plot.bar()` caused bars to use multiple colors instead of a single one (GH20585)

• Bug in validating color parameter caused extra color to be appended to the given color array. This happened to multiple plotting functions using matplotlib. (GH20726)
GroupBy/resample/rolling

- Bug in `pandas.core.window.Rolling.min()` and `pandas.core.window.Rolling.max()` with closed='left', a datetime-like index and only one entry in the series leading to segfault (GH24718)
- Bug in `pandas.core.groupby.GroupBy.first()` and `pandas.core.groupby.GroupBy.last()` with as_index=False leading to the loss of timezone information (GH15884)
- Bug in `DataFrame.resample()` when downsampling across a DST boundary (GH8531)
- Bug in `DataFrame.resample()` when resampling a weekly offset (GH9119, GH21459)
- Bug in `DataFrame.expanding()` in which the axis argument was not being respected during aggregations (GH23372)
- Bug preventing hash table creation with very large number (2^32) of rows (GH22805)
Reshaping

- Bug in `pandas.concat()` when joining resampled DataFrames with timezone aware index (GH13783)
- Bug in `pandas.concat()` when joining only Series the names argument of concat is no longer ignored (GH23490)
- Bug in `Series.combine_first()` with `datetime64[ns, tz]` dtype which would return tz-naive result (GH21469)
- Bug in `Series.where()` and `DataFrame.where()` with `datetime64[ns, tz]` dtype (GH21546)
- Bug in `DataFrame.where()` with an empty DataFrame and empty cond having non-bool dtype (GH21947)
- Bug in `Series.mask()` and `DataFrame.mask()` with list conditionals (GH21891)
- Bug in `DataFrame.replace()` raises RecursionError when converting OutOf Bounds `datetime64[ns, tz]` (GH20380)
- `pandas.core.groupby.GroupBy.rank()` now raises a `ValueError` when an invalid value is passed for argument `na_option` (GH22124)
- Bug in `get_dummies()` with Unicode attributes in Python 2 (GH22084)
- Bug in `DataFrame.replace()` raises RecursionError when replacing empty lists (GH22083)
- Bug in `Series.replace()` and `DataFrame.replace()` when dict is used as the `to_replace` value and one key in the dict is another key's value, the results were inconsistent between using integer key and using string key (GH20656)
- Bug in `DataFrame.drop_duplicates()` for empty DataFrame which incorrectly raises an error (GH20516)
- Bug in `pandas.wide_to_long()` when a string is passed to the stubnames argument and a column name is a substring of that stubname (GH22468)
- Bug in `merge()` when merging `datetime64[ns, tz]` data that contained a DST transition (GH18885)
- Bug in `merge_asof()` when merging on float values within defined tolerance (GH22981)
- Bug in `pandas.concat()` when concatenating a multicolumn DataFrame with tz-aware data against a DataFrame with a different number of columns (GH22796)
- Bug in `merge_asof()` where confusing error message raised when attempting to merge with missing values (GH23189)
- Bug in `DataFrame.nsmallest()` and `DataFrame.nlargest()` for dataframes that have a MultiIndex for columns (GH23033).
- Bug in `pandas.melt()` when passing column names that are not present in DataFrame (GH23575)
- Bug in `DataFrame.append()` with a `Series` with a dateutil timezone would raise a `TypeError` (GH23682)
- Bug in `Series` construction when passing no data and `dtype=string` (GH22477)
- Bug in `cut()` with bins as an overlapping `IntervalIndex` where multiple bins were returned per item instead of raising a `ValueError` (GH23980)
- Bug in `pandas.concat()` when joining `Series` datetimetz with `Series` category would lose timezone (GH23816)
- Bug in `DataFrame.join()` when joining on partial MultiIndex would drop names (GH20452).
- `DataFrame.nlargest()` and `DataFrame.nsmallest()` now returns the correct n values when keep != ‘all’ also when tied on the first columns (GH22752)
• Constructing a DataFrame with an index argument that wasn’t already an instance of Index was broken (GH22227).
• Bug in DataFrame prevented list subclasses to be used to construction (GH21226)
• Bug in DataFrame.unstack() and DataFrame.pivot_table() returning a misleading error message when the resulting DataFrame has more elements than int32 can handle. Now, the error message is improved, pointing towards the actual problem (GH20601)
• Bug in DataFrame.unstack() where a ValueError was raised when unstacking timezone aware values (GH18338)
• Bug in DataFrame.stack() where timezone aware values were converted to timezone naive values (GH19420)
• Bug in merge_asof() where a TypeError was raised when by_col were timezone aware values (GH21184)
• Bug showing an incorrect shape when throwing error during DataFrame construction. (GH20742)

Sparse

• Updating a boolean, datetime, or timedelta column to be Sparse now works (GH22367)
• Bug in Series.to_sparse() with Series already holding sparse data not constructing properly (GH22389)
• Providing a sparse_index to the SparseArray constructor no longer defaults the na-value to np.nan for all dtypes. The correct na_value for data.dtype is now used.
• Bug in SparseArray.nbytes under-reporting its memory usage by not including the size of its sparse index.
• Improved performance of Series.shift() for non-NA fill_value, as values are no longer converted to a dense array.
• Bug in DataFrame.groupby not including fill_value in the groups for non-NA fill_value when grouping by a sparse column (GH5078)
• Bug in unary inversion operator (~) on a SparseSeries with boolean values. The performance of this has also been improved (GH22835)
• Bug in SparseArray.unique() not returning the unique values (GH19595)
• Bug in SparseArray.nonzero() and SparseDataFrame.dropna() returning shifted/incorrect results (GH21172)
• Bug in DataFrame.apply() where dtypes would lose sparseness (GH23744)
• Bug in concat() when concatenating a list of Series with all-sparse values changing the fill_value and converting to a dense Series (GH24371)

Style

• background_gradient() now takes a text_color_threshold parameter to automatically lighten the text color based on the luminance of the background color. This improves readability with dark background colors without the need to limit the background colormap range. (GH21258)
• background_gradient() now also supports tablewise application (in addition to rowwise and columnwise) with axis=None (GH15204)
• bar() now also supports tablewise application (in addition to rowwise and columnwise) with axis=None and setting clipping range with vmin and vmax (GH21548 and GH21526). NaN values are also handled properly.
Build changes

- Building pandas for development now requires cython $\geq 0.28.2$ (GH21688)
- Testing pandas now requires hypothesis $\geq 3.58$. You can find the Hypothesis docs here, and a pandas-specific introduction in the contributing guide. (GH22280)
- Building pandas on macOS now targets minimum macOS 10.9 if run on macOS 10.9 or above (GH23424)

Other

- Bug where C variables were declared with external linkage causing import errors if certain other C libraries were imported before pandas. (GH24113)

Contributors

A total of 337 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- AJ Dyka +
- AJ Pryor, Ph.D +
- Aaron Critchley
- Adam Hooper
- Adam J. Stewart
- Adam Kim
- Adam Klimont +
- Addison Lynch +
- Alan Hogue +
- Alex Radu +
- Alex Rychyk
- Alex Strick van Linschoten +
- Alex Volkov +
- Alexander Buchkovsky
- Alexander Hess +
- Alexander Ponomaroff +
- Allison Browne +
- Aly Sivji
- Andrew
- Andrew Gross +
- Andrew Spott +
- Andy +
- Aniket uttam +
• Anjali2019 +
• Anjana S +
• Antti Kaihola +
• Anudeep Tubati +
• Arjun Sharma +
• Armin Varshokar
• Artem Bogachev
• ArtinSarraf +
• Barry Fitzgerald +
• Bart Aelterman +
• Ben James +
• Ben Nelson +
• Benjamin Grove +
• Benjamin Rowell +
• Benoit Paquet +
• Boris Lau +
• Brett Naul
• Brian Choi +
• C.A.M. Gerlach +
• Carl Johan +
• Chalmer Lowe
• Chang She
• Charles David +
• Cheuk Ting Ho
• Chris
• Chris Roberts +
• Christopher Whelan
• Chu Qing Hao +
• Da Cheezy Mobsta +
• Damini Satya
• Daniel Himmelstein
• Daniel Saxton +
• Darcy Meyer +
• DataOmbudsman
• David Arcos
• David Krych
• Dean Langsam +
• Diego Argueta +
• Diego Torres +
• Dobatymo +
• Doug Latornell +
• Dr. Irv
• Dylan Dmitri Gray +
• Eric Boxer +
• Eric Chea
• Erik +
• Erik Nilsson +
• Fabian Haase +
• Fabian Retkowski
• Fabien Aulaire +
• Fakabbir Amin +
• Fei Phoon +
• Fernando Margueirat +
• Florian Müller +
• Fábio Rosado +
• Gabe Fernando
• Gabriel Reid +
• Giftlin Rajaiah
• Gioia Ballin +
• Gjelt
• Gosuke Shibahara +
• Graham Inggs
• Guillaume Gay
• Guillaume Lemaitre +
• Hannah Ferchland
• Haochen Wu
• Hubert +
• HubertKl +
• HyunTruth +
• Iain Barr
• Ignacio Vergara Kausel +
• Irv Lustig +
pandas: powerful Python data analysis toolkit, Release 1.4.2

• IsvenC +
• Jacopo Rota
• Jakob Jarmar +
• James Bourbeau +
• James Myatt +
• James Winegar +
• Jan Rudolph
• Jared Groves +
• Jason Kiley +
• Javad Noorbakhsh +
• Jay Offerdahl +
• Jeff Reback
• Jeongmin Yu +
• Jeremy Schendel
• Jerod Estapa +
• Jesper Dramsch +
• Jim Jeon +
• Joe Jevnik
• Joel Nothman
• Joel Ostblom +
• Jordi Contestí
• Jorge López Fueyo +
• Joris Van den Bossche
• Jose Quinones +
• Jose Rivera-Rubio +
• Josh
• Jun +
• Justin Zheng +
• Kaiqi Dong +
• Kalyan Gokhale
• Kang Yoosam +
• Karl Dunkle Werner +
• Karmany Aggarwal +
• Kevin Markham +
• Kevin Sheppard
• Kimi Li +
• Koustav Samaddar +
• Krishna +
• Kristian Holsheimer +
• Ksenia Gueletina +
• Kyle Prestel +
• LJ +
• LeakedMemory +
• Li Jin +
• Licht Takeuchi
• Luca Donini +
• Luciano Viola +
• Mak Sze Chun +
• Marc Garcia
• Marius Potgieter +
• Mark Sikora +
• Markus Meier +
• Marlene Silva Marchena +
• Martin Babka +
• MatanCohe +
• Mateusz Woś +
• Mathew Topper +
• Matt Boggess +
• Matt Cooper +
• Matt Williams +
• Matthew Gilbert
• Matthew Roeschke
• Max Kanter
• Michael Odintsov
• Michael Silverstein +
• Michael-J-Ward +
• Mickaël Schoentgen +
• Miguel Sánchez de León Peque +
• Ming Li
• Mitar
• Mitch Negus
• Monson Shao +
• Moonsoo Kim +
• Mortada Mehyar
• Myles Braithwaite
• Nehil Jain +
• Nicholas Musolino +
• Nicolas Dickreuter +
• Nikhil Kumar Mengani +
• Nikoleta Glynatsi +
• Ondrej Kokes
• Pablo Ambrosio +
• Pamela Wu +
• Parfait G +
• Patrick Park +
• Paul
• Paul Ganssle
• Paul Reidy
• Paul van Mulbregt +
• Phillip Cloud
• Pietro Battiston
• Piyush Aggarwal +
• Prabakaran Kumareshan +
• Pulkit Maloo
• Pyry Kovanen
• Rajib Mitra +
• Redonnet Louis +
• Rhys Parry +
• Rick +
• Robin
• Roei.r +
• Romain Sa +
• Roman Imankulov +
• Roman Yurchak +
• Ruijing Li +
• Ryan +
• Ryan Nazareth +
• Rüdiger Busche +
• SEUNG HOON, SHIN +
• Sandrine Pataut +
• Sangwoong Yoon
• Santosh Kumar +
• Saurav Chakravorty +
• Scott McAllister +
• Sean Chan +
• Shadi Akiki +
• Shengpu Tang +
• Shirish Kadam +
• Simon Hawkins +
• Simon Riddell +
• Simone Basso
• Sinhrks
• Soyoun(Rose) Kim +
• Srinivas Reddy Thatiparthy ( ) +
• Stefaan Lippens +
• Stefano Cianciulli
• Stefano Miccoli +
• Stephen Childs
• Stephen Pascoe
• Steve Baker +
• Steve Cook +
• Steve Dower +
• Stéphan Taljaard +
• Sumin Byeon +
• Sören +
• Tamas Nagy +
• Tanya Jain +
• Tarbo Fukazawa
• Thein Oo +
• Thiago Cordeiro da Fonseca +
• Thierry Moisan
• Thiviyan Thanapalasingam +
• Thomas Lentali +
• Tim D. Smith +
• Tim Swast
• Tom Augspurger
• Tomasz Kluczkowski +
• Tony Tao +
• Triple0 +
• Troels Nielsen +
• Tuhin Mahmud +
• Tyler Reddy +
• Uddeshya Singh
• Uwe L. Korn +
• Vadym Bara +
• Varad Gunjal +
• Victor Maryama +
• Victor Villas
• Vincent La
• Vitória Helena +
• Vu Le
• Vyom Jain +
• Weiwen Gu +
• Wenhuan
• Wes Turner
• Wil Tan +
• William Ayd
• Yeojin Kim +
• Yitzhak Andrade +
• Yuecheng Wu +
• Yuliya Dovzhenko +
• Yury Bayda +
• Zac Hatfield-Dodds +
• aberres +
• aeltanawy +
• alilcchau +
• alimcmaster1
• alphaCTzo7G +
• amphy +
• araraonline +
• azure-pipelines[bot] +
• benarthur91 +
• bk521234 +
• cgangwar11 +
• chris-b1
• cxl923cc +
• dahlbaek +
• dannyhyunkim +
• darke-spirits +
• david-liu-brattle-1
• davidmvalente +
• deflatSOCO
• doosik_bae +
• dylanchase +
• eduardo naufel schettino +
• euri10 +
• evangelineliu +
• fengyqf +
• fjdi0d
• fl4p +
• fleimgruber +
• gfyoung
• h-vetinari
• harisbal +
• henriqueribeiro +
• himanshu awasthi
• hongshaoyang +
• igorfassen +
• jalazbe +
• jbrockmendel
• jh-wu +
• justinchann23 +
• louispotok
• marcosrullan +
• miker985
• nicolab100 +
5.8 Version 0.23

5.8.1 What’s new in 0.23.4 (August 3, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See Dropping Python 2.7 for more.
What’s new in v0.23.4

- **Fixed regressions**
- **Bug fixes**
- **Contributors**

**Fixed regressions**

- Python 3.7 with Windows gave all missing values for rolling variance calculations (GH21813)

**Bug fixes**

**Groupby/resample/rolling**

- Bug where calling `DataFrameGroupBy.agg()` with a list of functions including `ohlc` as the non-initial element would raise a `ValueError` (GH21716)
- Bug in `roll_quantile` caused a memory leak when calling `.rolling(...).quantile(q)` with q in (0,1) (GH21965)

**Missing**

- Bug in `Series.clip()` and `DataFrame.clip()` cannot accept list-like threshold containing `NaN` (GH19992)

**Contributors**

A total of 6 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Jeff Reback
- MeeseeksMachine +
- Tom Augspurger
- chris-b1
- h-vetinari
- meeseeksdev[bot]

**5.8.2 What’s new in 0.23.3 (July 7, 2018)**

This release fixes a build issue with the sdist for Python 3.7 (GH21785) There are no other changes.
Contributors

A total of 2 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Tom Augspurger
- meeseeksdev[bot] +

5.8.3 What’s new in 0.23.2 (July 5, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Note: pandas 0.23.2 is first pandas release that’s compatible with Python 3.7 (GH20552)

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See Dropping Python 2.7 for more.

What’s new in v0.23.2

- Logical reductions over entire DataFrame
- Fixed regressions
- Build changes
- Bug fixes
- Contributors

Logical reductions over entire DataFrame

`DataFrame.all()` and `DataFrame.any()` now accept `axis=None` to reduce over all axes to a scalar (GH19976)

```python
In [1]: df = pd.DataFrame({"A": [1, 2], "B": [True, False]})
In [2]: df.all(axis=None)
Out[2]: False
```

This also provides compatibility with NumPy 1.15, which now dispatches to `DataFrame.all`. With NumPy 1.15 and pandas 0.23.1 or earlier, `numpy.all()` will no longer reduce over every axis:

```bash
>>> # NumPy 1.15, pandas 0.23.1
>>> np.any(pd.DataFrame({"A": [False], "B": [False]}))
A  False
B  False
dtype: bool
```

With pandas 0.23.2, that will correctly return False, as it did with NumPy < 1.15.
In [3]: np.any(pd.DataFrame({"A": [False], "B": [False]}))
Out[3]: False

Fixed regressions

- Fixed regression in `to_csv()` when handling file-like object incorrectly (GH21471)
- Re-allowed duplicate level names of a MultiIndex. Accessing a level that has a duplicate name by name still raises an error (GH19029).
- Bug in both `DataFrame.first_valid_index()` and `Series.first_valid_index()` raised for a row index having duplicate values (GH21441)
- Fixed printing of DataFrames with hierarchical columns with long names (GH21180)
- Fixed regression in `reindex()` and `groupby()` with a MultiIndex or multiple keys that contains categorical datetime-like values (GH21390).
- Fixed regression in unary negative operations with object dtype (GH21380)
- Bug in `Timestamp.ceil()` and `Timestamp.floor()` when timestamp is a multiple of the rounding frequency (GH21262)
- Fixed regression in `to_clipboard()` that defaulted to copying dataframes with space delimited instead of tab delimited (GH21104)

Build changes

- The source and binary distributions no longer include test data files, resulting in smaller download sizes. Tests relying on these data files will be skipped when using `pandas.test()` (GH19320)

Bug fixes

Conversion

- Bug in constructing `Index` with an iterator or generator (GH21470)
- Bug in `Series.nlargest()` for signed and unsigned integer dtypes when the minimum value is present (GH21426)

Indexing

- Bug in `Index.get_indexer_non_unique()` with categorical key (GH21448)
- Bug in comparison operations for `MultiIndex` where error was raised on equality / inequality comparison involving a MultiIndex with `nlevels == 1` (GH21149)
- Bug in `DataFrame.drop()` behaviour is not consistent for unique and non-unique indexes (GH21494)
- Bug in `DataFrame.duplicated()` with a large number of columns causing a 'maximum recursion depth exceeded' (GH21524)

I/O

- Bug in `read_csv()` that caused it to incorrectly raise an error when `nrows=0, low_memory=True, and index_col` was not `None` (GH21141)
- Bug in `json_normalize()` when formatting the `record_prefix` with integer columns (GH21536)

Categorical
• Bug in rendering *Series* with *Categorical* dtype in rare conditions under Python 2.7 (GH21002)

**Timezones**

• Bug in *Timestamp* and *DatetimeIndex* where passing a *Timestamp* localized after a DST transition would return a datetime before the DST transition (GH20854)

• Bug in comparing *DataFrame* with tz-aware *DatetimeIndex* columns with a DST transition that raised a *KeyError* (GH19970)

• Bug in *Datet imeIndex.shift()* where an *AssertionError* would raise when shifting across DST (GH8616)

• Bug in *Timestamp* constructor where passing an invalid timezone offset designator (Z) would not raise a *ValueError* (GH8910)

• Bug in *Timestamp.replace()* where replacing at a DST boundary would retain an incorrect offset (GH7825)

• Bug in *Datet imeIndex.reindex()* when reindexing a tz-naive and tz-aware *Datet imeIndex* (GH8306)

• Bug in *Datet imeIndex.resample()* when downsampling across a DST boundary (GH8531)

**Timedelta**

• Bug in *Timedelta* where non-zero timedeltas shorter than 1 microsecond were considered False (GH21484)

**Contributors**

A total of 17 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• David Krych
• Jacopo Rota +
• Jeff Reback
• Jeremy Schendel
• Joris Van den Bossche
• Kalyan Gokhale
• Matthew Roeschke
• Michael Odintsov +
• Ming Li
• Pietro Battiston
• Tom Augspurger
• Uddeshya Singh
• Vu Le +
• alimcmaster1 +
• david-liu-brattle-1 +
• gfy oun g
• jbrockmendel
5.8.4 What’s new in 0.23.1 (June 12, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See Dropping Python 2.7 for more.

What’s new in v0.23.1

- Fixed regressions
- Performance improvements
- Bug fixes
- Contributors

Fixed regressions

Comparing Series with datetime.date

We’ve reverted a 0.23.0 change to comparing a Series holding datetimes and a datetime.date object (GH21152). In pandas 0.22 and earlier, comparing a Series holding datetimes and datetime.date objects would coerce the datetime.date to a datetime before comparing. This was inconsistent with Python, NumPy, and DatetimeIndex, which never consider a datetime and datetime.date equal.

In 0.23.0, we unified operations between DatetimeIndex and Series, and in the process changed comparisons between a Series of datetimes and datetime.date without warning.

We’ve temporarily restored the 0.22.0 behavior, so datetimes and dates may again compare equal, but restore the 0.23.0 behavior in a future release.

To summarize, here’s the behavior in 0.22.0, 0.23.0, 0.23.1:

```python
# 0.22.0... Silently coerce the datetime.date
>>> import datetime
datetime

>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0   True
1  False
dtype: bool

# 0.23.0... Do not coerce the datetime.date
>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0   False
1  False
dtype: bool

# 0.23.1... Coerce the datetime.date with a warning
>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
/bin/python:1: FutureWarning: Comparing Series of datetimes with 'datetime.date'. ␣ Currently, the 'datetime.date' is coerced to a datetime. In the future pandas will
```

(continues on next page)
not coerce, and the values not compare equal to the 'datetime.date'.

To retain the current behavior, convert the 'datetime.date' to a
datetime with 'pd.Timestamp'.

```python
#!/bin/python3
0 True
1 False
dtype: bool
```

In addition, ordering comparisons will raise a TypeError in the future.

**Other fixes**

- Reverted the ability of `to_sql()` to perform multivalue inserts as this caused regression in certain cases (GH21103). In the future this will be made configurable.

- Fixed regression in the `DatetimeIndex.date` and `DatetimeIndex.time` attributes in case of timezone-aware data: `DatetimeIndex.time` returned a tz-aware time instead of tz-naive (GH21267) and `DatetimeIndex.date` returned incorrect date when the input date has a non-UTC timezone (GH21230).

- Fixed regression in `pandas.io.json.json_normalize()` when called with `None` values in nested levels in JSON, and to not drop keys with value as `None` (GH21158, GH21356).

- Bug in `to_csv()` causes encoding error when compression and encoding are specified (GH21241, GH21118)

- Bug preventing pandas from being importable with -OO optimization (GH21071)

- Bug in `Categorical.fillna()` incorrectly raising a TypeError when value the individual categories are iterable and value is an iterable (GH21097, GH19788)

- Fixed regression in constructors coercing NA values like `None` to strings when passing `dtype=str` (GH21083)

- Regression in `pivot_table()` where an ordered `Categorical` with missing values for the pivot's index would give a mis-aligned result (GH21133)

- Fixed regression in merging on boolean index/columns (GH21119).

**Performance improvements**

- Improved performance of `CategoricalIndex.is_monotonic_increasing()`, `CategoricalIndex.is_monotonic_decreasing()` and `CategoricalIndex.is_monotonic()` (GH21025)

- Improved performance of `CategoricalIndex.is_unique()` (GH21107)

**Bug fixes**

**Groupby/resample/rolling**

- Bug in `DataFrame.agg()` where applying multiple aggregation functions to a `DataFrame` with duplicated column names would cause a stack overflow (GH21063)

- Bug in `pandas.core.groupby.GroupBy.ffill()` and `pandas.core.groupby.GroupBy.bfill()` where the fill within a grouping would not always be applied as intended due to the implementations' use of a non-stable sort (GH21207)

- Bug in `pandas.core.groupby.GroupBy.rank()` where results did not scale to 100% when specifying `method='dense'` and `pct=True`

- Bug in `pandas.DataFrame.rolling()` and `pandas.Series.rolling()` which incorrectly accepted a 0 window size rather than raising (GH21286)
Data-type specific

- Bug in `Series.str.replace()` where the method throws `TypeError` on Python 3.5.2 (GH21078)
- Bug in `Timedelta` where passing a float with a unit would prematurely round the float precision (GH14156)
- Bug in `pandas.testing.assert_index_equal()` which raised `AssertionError` incorrectly, when comparing two `CategoricalIndex` objects with param `check_categorical=False` (GH19776)

Sparse

- Bug in `SparseArray.shape` which previously only returned the shape `SparseArray.sp_values` (GH21126)

Indexing

- Bug in `Series.reset_index()` where appropriate error was not raised with an invalid level name (GH20925)
- Bug in `interval_range()` when `start/periods` or `end/periods` are specified with float `start` or `end` (GH21161)
- Bug in `MultiIndex.set_names()` where error raised for a `MultiIndex` with `nlevels == 1` (GH21149)
- Bug in `IntervalIndex` constructors where creating an `IntervalIndex` from categorical data was not fully supported (GH21243, GH21253)
- Bug in `MultiIndex.sort_index()` which was not guaranteed to sort correctly with `level=1`; this was also causing data misalignment in particular `DataFrame.stack()` operations (GH20994, GH20945, GH21052)

Plotting

- New keywords (sharex, sharey) to turn on/off sharing of x/y-axis by subplots generated with `pandas.DataFrame().groupby().boxplot()` (GH20968)

I/O

- Bug in IO methods specifying `compression='zip'` which produced uncompressed zip archives (GH17778, GH21144)
- Bug in `DataFrame.to_stata()` which prevented exporting DataFrames to buffers and most file-like objects (GH21041)
- Bug in `read_stata()` and StataReader which did not correctly decode utf-8 strings on Python 3 from Stata 14 files (dta version 118) (GH21244)
- Bug in IO JSON `read_json()` reading empty JSON schema with `orient='table'` back to `DataFrame` caused an error (GH21287)

Reshaping

- Bug in `concat()` where error was raised in concatenating `Series` with numpy scalar and tuple names (GH21015)
- Bug in `concat()` warning message providing the wrong guidance for future behavior (GH21101)

Other

- Tab completion on `Index` in IPython no longer outputs deprecation warnings (GH21125)
- Bug preventing pandas being used on Windows without C++ redistributable installed (GH21106)
Contributors

A total of 30 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam J. Stewart
- Adam Kim +
- Aly Sivji
- Chalmer Lowe +
- Damini Satya +
- Dr. Irv
- Gabe Fernando +
- Giftlin Rajaiah
- Jeff Reback
- Jeremy Schendel +
- Joris Van den Bossche
- Kalyan Gokhale +
- Kevin Sheppard
- Matthew Roeschke
- Max Kanter +
- Ming Li
- Pyry Kovanen +
- Stefano Cianciulli
- Tom Augspurger
- Uddeshya Singh +
- Wenhuan
- William Ayd
- chris-b1
- gfyoungh
- h-vetinari
- np4rd +
- ssikdar1 +
- tmnhat2001
- topper-123
- zertrin +
5.8.5 What’s new in 0.23.0 (May 15, 2018)

This is a major release from 0.22.0 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Round-trippable JSON format with ‘table’ orient.
- Instantiation from dicts respects order for Python 3.6+.
- Dependent column arguments for assign.
- Merging / sorting on a combination of columns and index levels.
- Extending pandas with custom types.
- Excluding unobserved categories from groupby.
- Changes to make output shape of DataFrame.apply consistent.

Check the API Changes and deprecations before updating.

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See Dropping Python 2.7 for more.

What’s new in v0.23.0

- New features
  - JSON read/write round-trippable with orient='table'
  - Method .assign() accepts dependent arguments
  - Merging on a combination of columns and index levels
  - Sorting by a combination of columns and index levels
  - Extending pandas with custom types (experimental)
  - New observed keyword for excluding unobserved categories in GroupBy
  - Rolling/Expanding.apply() accepts raw=False to pass a Series to the function
  - DataFrame.interpolate has gained the limit_area kwarg
  - Function get_dummies now supports dtype argument
  - Timedelta mod method
  - Method .rank() handles inf values when NaN are present
  - Series.str.cat has gained the join kwarg
  - DataFrame.astype performs column-wise conversion to Categorical
  - Other enhancements
- Backwards incompatible API changes
  - Dependencies have increased minimum versions
  - Instantiation from dicts preserves dict insertion order for Python 3.6+
- Deprecate Panel
- pandas.core.common removals
- Changes to make output of DataFrame.apply consistent
- Concatenation will no longer sort
- Build changes
- Index division by zero fills correctly
- Extraction of matching patterns from strings
- Default value for the ordered parameter of CategoricalDtype
- Better pretty-printing of DataFrames in a terminal
- Datetimelike API changes
- Other API changes

• Deprecations
• Removal of prior version deprecations/changes
• Performance improvements
• Documentation changes
• Bug fixes
  - Categorical
  - Datetimelike
  - Timedelta
  - Timezones
  - Offsets
  - Numeric
  - Strings
  - Indexing
  - MultiIndex
  - IO
  - Plotting
  - GroupBy/resample/rolling
  - Sparse
  - Reshaping
  - Other

• Contributors
New features

**JSON read/write round-trippable with orient=’table’**

A DataFrame can now be written to and subsequently read back via JSON while preserving metadata through usage of the orient=’table’ argument (see GH18912 and GH9146). Previously, none of the available orient values guaranteed the preservation of dtypes and index names, amongst other metadata.

```python
In [1]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
                      'bar': ['a', 'b', 'c', 'd'],
                      'baz': pd.date_range('2018-01-01', freq='d', periods=4),
                      'qux': pd.Categorical(['a', 'b', 'c', 'c'])},
                      index=pd.Index(range(4), name='idx'))

In [2]: df
Out[2]:
   foo  bar     baz          qux
   idx
0     1   a 2018-01-01     a
1     2   b 2018-01-02     b
2     3   c 2018-01-03     c
3     4   d 2018-01-04     c

[4 rows x 4 columns]

In [3]: df.dtypes
Out[3]:
foo     int64
bar    object
baz  datetime64[ns]
qux   category

Length: 4, dtype: object

In [4]: df.to_json('test.json', orient='table')

In [5]: new_df = pd.read_json('test.json', orient='table')

In [6]: new_df
Out[6]:
   foo  bar     baz          qux
   idx
0     1   a 2018-01-01     a
1     2   b 2018-01-02     b
2     3   c 2018-01-03     c
3     4   d 2018-01-04     c

[4 rows x 4 columns]

In [7]: new_df.dtypes
Out[7]:
foo     int64
bar    object
```

(continues on next page)
Please note that the string index is not supported with the round trip format, as it is used by default in write_json to indicate a missing index name.

In [8]: df.index.name = 'index'

In [9]: df.to_json('test.json', orient='table')

In [10]: new_df = pd.read_json('test.json', orient='table')

In [11]: new_df
Out[11]:
   foo  bar           baz  qux
0  1.0  a  2018-01-01  a
1  2.0  b  2018-01-02  b
2  3.0  c  2018-01-03  c
3  4.0  d  2018-01-04  c

[4 rows x 4 columns]

In [12]: new_df.dtypes
Out[12]:
   foo     object
   bar     object
   baz  datetime64[ns]
   qux     category

Method .assign() accepts dependent arguments

The DataFrame.assign() now accepts dependent keyword arguments for python version later than 3.6 (see also PEP 468). Later keyword arguments may now refer to earlier ones if the argument is a callable. See the documentation here (GH14207)

In [13]: df = pd.DataFrame({'A': [1, 2, 3]})

In [14]: df
Out[14]:
   A
0  1
1  2
2  3

[3 rows x 1 columns]

In [15]: df.assign(B=df.A, C=lambda x: x['A'] + x['B'])
Out[15]:
   A  B  C
0  1  2  3
1  2  3  5
2  3  4  7

[3 rows x 3 columns]
Warning: This may subtly change the behavior of your code when you’re using `.assign()` to update an existing column. Previously, callables referring to other variables being updated would get the “old” values.

Previous behavior:

```python
In [2]: df = pd.DataFrame({"A": [1, 2, 3]})
In [3]: df.assign(A=lambda df: df.A + 1, C=lambda df: df.A * -1)
Out[3]:
   A  C
0  2 -1
1  3 -2
2  4 -3
```

New behavior:

```python
In [16]: df.assign(A=df.A + 1, C=lambda df: df.A * -1)
Out[16]:
   A  C
0  2 -2
1  3 -3
2  4 -4
```

Merging on a combination of columns and index levels

Strings passed to `DataFrame.merge()` as the `on`, `left_on`, and `right_on` parameters may now refer to either column names or index level names. This enables merging DataFrame instances on a combination of index levels and columns without resetting indexes. See the `Merge on columns and levels` documentation section. (GH14355)

```python
In [17]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [18]: left = pd.DataFrame({
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3'],
                     'key2': ["K0", "K1", "K0", "K1"],
                     index=left_index})

In [19]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [20]: right = pd.DataFrame({
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3'],
                    'key2': ["K0", "K0", "K0", "K1"],
                    index=right_index})
```
In [21]: left.merge(right, on=['key1', 'key2'])
Out[21]:
A    B  key2  C  D
key1
K0  A0  B0  K0  C0  D0
K1  A2  B2  K0  C1  D1
K2  A3  B3  K1  C3  D3
[3 rows x 5 columns]

Sorting by a combination of columns and index levels

Strings passed to DataFrame.sort_values() as the by parameter may now refer to either column names or index level names. This enables sorting DataFrame instances by a combination of index levels and columns without resetting indexes. See the Sorting by Indexes and Values documentation section. (GH14353)

# Build MultiIndex
In [22]: idx = pd.MultiIndex.from_tuples([(a, 1), (a, 2), (a, 2),
                                  (b, 2), (b, 1), (b, 1)])

In [23]: idx.names = ['first', 'second']

# Build DataFrame
In [24]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
                           index=idx)

In [25]: df_multi
Out[25]:
   A
 first  second
a      1      6
  2      5
  2      4
b      2      3
  1      2
  1      1

[6 rows x 1 columns]

# Sort by 'second' (index) and 'A' (column)
In [26]: df_multi.sort_values(by=['second', 'A'])
Out[26]:
   A
 first  second
b  1      1
  1      2
a  1      6
Extending pandas with custom types (experimental)

pandas now supports storing array-like objects that aren’t necessarily 1-D NumPy arrays as columns in a DataFrame or values in a Series. This allows third-party libraries to implement extensions to NumPy’s types, similar to how pandas implemented categoricals, datetimes with timezones, periods, and intervals.

As a demonstration, we’ll use cyberpandas, which provides an IPArray type for storing ip addresses.

```
In [1]: from cyberpandas import IPArray

In [2]: values = IPArray([...
                     ...:     0,
                     ...:     3232235777,
                     ...:     42540766452641154071740215577757643572
                     ...:     ])

IPArray isn't a normal 1-D NumPy array, but because it's a pandas ExtensionArray, it can be stored properly inside pandas' containers.

In [3]: ser = pd.Series(values)

In [4]: ser
Out[4]:
0    0.0.0.0
1  192.168.1.1
2 2001:db8:85a3::8a2e:370:7334
dtype: ip

Notice that the dtype is ip. The missing value semantics of the underlying array are respected:

```
In [5]: ser.isna()
Out[5]:
0   True
1  False
2  False
dtype: bool
```

For more, see the extension types documentation. If you build an extension array, publicize it on our ecosystem page.
New observed keyword for excluding unobserved categories in GroupBy

Grouping by a categorical includes the unobserved categories in the output. When grouping by multiple categorical columns, this means you get the cartesian product of all the categories, including combinations where there are no observations, which can result in a large number of groups. We have added a keyword `observed` to control this behavior, it defaults to `observed=False` for backward-compatibility. (GH14942, GH8138, GH15217, GH17594, GH8669, GH20583, GH20902)

```
In [27]: cat1 = pd.Categorical(
    ....:     ['a', 'a', 'b', 'b'],
    ....:     categories=['a', 'b', 'z'], ordered=True)
    ....:

In [28]: cat2 = pd.Categorical(
    ....:     ['c', 'd', 'c', 'd'],
    ....:     categories=['c', 'd', 'y'], ordered=True)
    ....:

In [29]: df = pd.DataFrame({'A': cat1, 'B': cat2, 'values': [1, 2, 3, 4]})

In [30]: df['C'] = ['foo', 'bar'] * 2

In [31]: df
Out[31]:
     A  B     values  C  
0    a  c       foo  foo
1    a  d       bar  bar
2    b  c       foo  foo
3    b  d       bar  bar

[4 rows x 4 columns]
```

To show all values, the previous behavior:

```
In [32]: df.groupby(['A', 'B', 'C'], observed=False).count()
Out[32]:
   values
   A   B   C
a  c    bar  0
     foo  1
d  bar  1
     foo  0
     y    bar  0
     ...  ...
z  c    foo  0
d  bar  0
     foo  0
     y    bar  0
     foo  0

[18 rows x 1 columns]
```

To show only observed values:

```
In [33]: df.groupby(['A', 'B', 'C'], observed=True).count()
(continues on next page)
```
For pivoting operations, this behavior is already controlled by the dropna keyword:

```python
In [34]: cat1 = pd.Categorical(["a", "a", "b", "b"],
                   categories=["a", "b", "z"], ordered=True)
In [35]: cat2 = pd.Categorical(["c", "d", "c", "d"],
                   categories=["c", "d", "y"], ordered=True)
In [36]: df = pd.DataFrame({"A": cat1, "B": cat2, "values": [1, 2, 3, 4]})
```

```python
In [37]: df
Out[37]:
   A  B  values
0  a  c     1
1  a  d     2
2  b  c     3
3  b  d     4
[4 rows x 3 columns]
```

```python
In [38]: pd.pivot_table(df, values='values', index=['A', 'B'],
                   dropna=True)
Out[38]:
   values
  A B
a c 1
a d 2
b c 3
b d 4
[4 rows x 1 columns]
```

```python
In [39]: pd.pivot_table(df, values='values', index=['A', 'B'],
                   dropna=False)
Out[39]:
   values
  A B
a c 1.0
```

(continues on next page)
Rolling/Expanding.apply() accepts `raw=False` to pass a Series to the function

Series.rolling().apply(), DataFrame.rolling().apply(), Series.expanding().apply(), and DataFrame.expanding().apply() have gained a `raw=None` parameter. This is similar to DataFrame.apply(). This parameter, if True allows one to send a np.ndarray to the applied function. If False a Series will be passed. The default is None, which preserves backward compatibility, so this will default to True, sending an np.ndarray. In a future version the default will be changed to False, sending a Series. (GH5071, GH20584)

```python
In [40]: s = pd.Series(np.arange(5), np.arange(5) + 1)

In [41]: s
Out[41]:
1   0
2   1
3   2
4   3
5   4
Length: 5, dtype: int64
```

Pass a Series:

```python
In [42]: s.rolling(2, min_periods=1).apply(lambda x: x.iloc[-1], raw=False)
Out[42]:
1   0.0
2   1.0
3   2.0
4   3.0
5   4.0
Length: 5, dtype: float64
```

Mimic the original behavior of passing a ndarray:

```python
In [43]: s.rolling(2, min_periods=1).apply(lambda x: x[-1], raw=True)
Out[43]:
1   0.0
2   1.0
3   2.0
4   3.0
5   4.0
Length: 5, dtype: float64
```
**DataFrame.interpolate has gained the limit_area kwarg**

*DataFrame.interpolate()* has gained a *limit_area* parameter to allow further control of which NaNs are replaced. Use *limit_area='inside'* to fill only NaNs surrounded by valid values or use *limit_area='outside'* to fill only NaNs outside the existing valid values while preserving those inside. (GH16284) See the full documentation here.

```python
In [44]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan,
          ....:               np.nan, 13, np.nan, np.nan])

In [45]: ser
Out[45]:
0     NaN
1     NaN
2    5.0
3     NaN
4     NaN
5     NaN
6   13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill one consecutive inside value in both directions

```python
In [46]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
Out[46]:
0     NaN
1     NaN
2    5.0
3    7.0
4     NaN
5    11.0
6   13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill all consecutive outside values backward

```python
In [47]: ser.interpolate(limit_direction='backward', limit_area='outside')
Out[47]:
0    5.0
1    5.0
2    5.0
3     NaN
4     NaN
5     NaN
6   13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill all consecutive outside values in both directions
In [48]: ser.interpolate(limit_direction='both', limit_area='outside')
Out[48]:
0 5.0  
1 5.0  
2 5.0  
3 NaN 
4 NaN 
5 NaN 
6 13.0 
7 13.0 
8 13.0 
Length: 9, dtype: float64

Function get_dummies now supports dtype argument

The `get_dummies()` now accepts a `dtype` argument, which specifies a dtype for the new columns. The default remains `uint8`. (GH18330)

In [49]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [5, 6]})
In [50]: pd.get_dummies(df, columns=['c']).dtypes
Out[50]:
a   int64
b   int64
c_5  uint8
c_6  uint8
Length: 4, dtype: object
In [51]: pd.get_dummies(df, columns=['c'], dtype=bool).dtypes
Out[51]:
a   int64
b   int64
c_5  bool
c_6  bool
Length: 4, dtype: object

Timedelta mod method

mod (%) and divmod operations are now defined on Timedelta objects when operating with either timedelta-like or with numeric arguments. See the documentation here. (GH19365)

In [52]: td = pd.Timedelta(hours=37)
In [53]: td % pd.Timedelta(minutes=45)
Out[53]: Timedelta('0 days 00:15:00')
**Method .rank() handles inf values when NaN are present**

In previous versions, .rank() would assign inf elements NaN as their ranks. Now ranks are calculated properly. (GH6945)

```python
In [54]: s = pd.Series([-np.inf, 0, 1, np.nan, np.inf])
In [55]: s
Out[55]:
0   -inf
1     0
2     1
3      NaN
4      inf
Length: 5, dtype: float64
```

Previous behavior:

```python
In [11]: s.rank()
Out[11]:
0   1.0
1   2.0
2   3.0
3      NaN
4      NaN
dtype: float64
```

Current behavior:

```python
In [56]: s.rank()
Out[56]:
0   1.0
1   2.0
2   3.0
3      NaN
4   4.0
Length: 5, dtype: float64
```

Furthermore, previously if you rank inf or -inf values together with NaN values, the calculation won’t distinguish NaN from infinity when using ‘top’ or ‘bottom’ argument.

```python
In [57]: s = pd.Series([np.nan, np.nan, -np.inf, -np.inf])
In [58]: s
Out[58]:
0    NaN
1    NaN
2   -inf
3   -inf
Length: 4, dtype: float64
```

Previous behavior:
In [15]: s.rank(na_option='top')
Out[15]:
0   2.5
1   2.5
2   2.5
3   2.5
dtype: float64

Current behavior:

In [59]: s.rank(na_option='top')
Out[59]:
0   1.5
1   1.5
2   3.5
3   3.5
Length: 4, dtype: float64

These bugs were squashed:

- Bug in DataFrame.rank() and Series.rank() when method='dense' and pct=True in which percentile ranks were not being used with the number of distinct observations (GH15630)
- Bug in Series.rank() and DataFrame.rank() when ascending='False' failed to return correct ranks for infinity if NaN were present (GH19538)
- Bug in DataFrameGroupBy.rank() where ranks were incorrect when both infinity and NaN were present (GH20561)

Series.str.cat has gained the join kwarg

Previously, Series.str.cat() did not – in contrast to most of pandas – align Series on their index before concatenation (see GH18657). The method has now gained a keyword join to control the manner of alignment, see examples below and here.

In v.0.23 join will default to None (meaning no alignment), but this default will change to 'left' in a future version of pandas.

In [60]: s = pd.Series(['a', 'b', 'c', 'd'])
In [61]: t = pd.Series(['b', 'd', 'e', 'c'], index=[1, 3, 4, 2])
In [62]: s.str.cat(t)
Out[62]:
0  NaN
1  bb
2  cc
3  dd
Length: 4, dtype: object
In [63]: s.str.cat(t, join='left', na_rep='--')
Out[63]:
0  a--
1  bb
(continues on next page)
Furthermore, `Series.str.cat()` now works for `CategoricalIndex` as well (previously raised a `ValueError`; see GH20842).

**DataFrame.astype** performs column-wise conversion to `Categorical`

`DataFrame.astype()` can now perform column-wise conversion to `Categorical` by supplying the string 'category' or a `CategoricalDtype`. Previously, attempting this would raise a `NotImplementedError`. See the `Object creation` section of the documentation for more details and examples. (GH12860, GH18099)

Supplying the string 'category' performs column-wise conversion, with only labels appearing in a given column set as categories:

```
In [64]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd'))
In [65]: df = df.astype('category')
In [66]: df['A'].dtype
Out[66]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)
In [67]: df['B'].dtype
Out[67]: CategoricalDtype(categories=['b', 'c', 'd'], ordered=False)
```

Supplying a `CategoricalDtype` will make the categories in each column consistent with the supplied dtype:

```
In [68]: from pandas.api.types import CategoricalDtype
In [69]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd'))
In [70]: cdt = CategoricalDtype(categories=list('abcd'), ordered=True)
In [71]: df = df.astype(cdt)
In [72]: df['A'].dtype
Out[72]: CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)
In [73]: df['B'].dtype
Out[73]: CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)
```
Other enhancements

- Unary + now permitted for Series and DataFrame as numeric operator (GH16073)
- Better support for `to_excel()` output with the xlsxwriter engine. (GH16149)
- `pandas.tseries.frequencies.to_offset()` now accepts leading ‘+’ signs e.g. ‘+1h’. (GH18171)
- `MultiIndex.unique()` now supports the level= argument, to get unique values from a specific index level (GH17896)
- `pandas.ioformats.style.Styler` now has method `hide_index()` to determine whether the index will be rendered in output (GH14194)
- `pandas.ioformats.style.Styler` now has method `hide_columns()` to determine whether columns will be hidden in output (GH14194)
- Improved wording of `ValueError` raised in `to_datetime()` when unit= is passed with a non-convertible value (GH14350)
- `Series.fillna()` now accepts a Series or a dict as a value for a categorical dtype (GH17033)
- `pandas.read_clipboard()` updated to use qtpy, falling back to PyQt5 and then PyQt4, adding compatibility with Python3 and multiple python-qt bindings (GH17722)
- Improved wording of `ValueError` raised in `read_csv()` when the usecols argument cannot match all columns. (GH17301)
- `DataFrame.corrwith()` now silently drops non-numeric columns when passed a Series. Before, an exception was raised (GH18570).
- `IntervalIndex` now supports time zone aware `Interval` objects (GH18537, GH18538)
- `Series() / DataFrame()` tab completion also returns identifiers in the first level of a `MultiIndex()`. (GH16326)
- `read_excel()` has gained the `nrows` parameter (GH16645)
- `DataFrame.append()` can now in more cases preserve the type of the calling dataframe's columns (e.g. if both are CategoricalIndex) (GH18359)
- `DataFrame.to_json()` and `Series.to_json()` now accept an `index` argument which allows the user to exclude the index from the JSON output (GH17394)
- `IntervalIndex.to_tuples()` has gained the na_tuple parameter to control whether NA is returned as a tuple of NA, or NA itself (GH18756)
- `Categorical.rename_categories`, `CategoricalIndex.rename_categories` and `Series.cat.rename_categories` can now take a callable as their argument (GH18862)
- `Interval` and `IntervalIndex` have gained a `length` attribute (GH18789)
- Resampler objects now have a functioning `pipe` method. Previously, calls to pipe were diverted to the mean method (GH17905).
- `is_scalar()` now returns True for DateOffset objects (GH18943).
- `DataFrame.pivot()` now accepts a list for the values= kwarg (GH17160).
- Added `pandas.api.extensions.register_dataframe_accessor()`, `pandas.api.extensions.register_series_accessor()`, and `pandas.api.extensions.register_index_accessor()`, accessor for libraries downstream of pandas to register custom accessors like .cat on pandas objects. See Registering Custom Accessors for more (GH14781).
- `IntervalIndex.astype` now supports conversions between subtypes when passed an `IntervalDtype` (GH19197)
- `IntervalIndex` and its associated constructor methods (from_arrays, from_breaks, from_tuples) have gained a `dtype` parameter (GH19262)
- Added `pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing()` and `pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing()` (GH17015)
- For subclassed `DataFrames`, `DataFrame.apply()` will now preserve the `Series` subclass (if defined) when passing the data to the applied function (GH19822)
- `DataFrame.from_dict()` now accepts a `columns` argument that can be used to specify the column names when `orient='index'` is used (GH18529)
- Added option `display.html.use_mathjax` so MathJax can be disabled when rendering tables in Jupyter notebooks (GH19856, GH19824)
- `DataFrame.replace()` now supports the `method` parameter, which can be used to specify the replacement method when `to_replace` is a scalar, list or tuple and `value` is `None` (GH19632)
- `Timestamp.month_name()`, `DatetimeIndex.month_name()`, and `Series.dt.month_name()` are now available (GH12805)
- `Timestamp.day_name()` and `DatetimeIndex.day_name()` are now available to return day names with a specified locale (GH12806)
- `DataFrame.to_sql()` now performs a multi-value insert if the underlying connection supports it; rather than inserting row by row. SQLAlchemy dialects supporting multi-value inserts include: mysql, postgresql, sqlite and any dialect with `supports_multivalues_insert`. (GH14315, GH8953)
- `read_html()` now accepts a `displayed_only` keyword argument to controls whether or not hidden elements are parsed (True by default) (GH20027)
- `read_html()` now reads all `<tbody>` elements in a `<table>`, not just the first. (GH20690)
- `quantile()` and `quantile()` now accept the `interpolation` keyword, `linear` by default (GH20497)
- `zip` compression is supported via `compression=zip` in `DataFrame.to_pickle()`, `Series.to_pickle()`, `DataFrame.to_csv()`, `Series.to_csv()`, `DataFrame.to_json()`, `Series.to_json()`. (GH17778)
- `WeekOfMonth` constructor now supports `n=0` (GH20517).
- `DataFrame` and `Series` now support matrix multiplication (@) operator (GH10259) for Python>=3.5
- Updated `DataFrame.to_gbq()` and `pandas.read_gbq()` signature and documentation to reflect changes from the pandas-gbq library version 0.4.0. Adds intersphinx mapping to pandas-gbq library. (GH20564)
- Added new writer for exporting stata data files in version 117, `StataWriter117`. This format supports exporting strings with lengths up to 2,000,000 characters (GH16450)
- `to_hdf()` and `read_hdf()` now accept an `errors` keyword argument to control encoding error handling (GH20835)
- `cut()` has gained the `duplicates='raise' | 'drop'` option to control whether to raise on duplicated edges (GH20947)
- `date_range()`, `timedelta_range()`, and `interval_range()` now return a linearly spaced index if `start`, `stop`, and `periods` are specified, but `freq` is not. (GH20808, GH20983, GH20976)
Backwards incompatible API changes

Dependencies have increased minimum versions

We have updated our minimum supported versions of dependencies (GH15184). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>python-dateutil</td>
<td>2.5.0</td>
<td>X</td>
<td>GH15184</td>
</tr>
<tr>
<td>openpyxl</td>
<td>2.4.0</td>
<td></td>
<td>GH15184</td>
</tr>
<tr>
<td>beautifulsoup4</td>
<td>4.2.1</td>
<td></td>
<td>GH20082</td>
</tr>
<tr>
<td>setuptools</td>
<td>24.2.0</td>
<td></td>
<td>GH20698</td>
</tr>
</tbody>
</table>

Instantiation from dicts preserves dict insertion order for Python 3.6+

Until Python 3.6, dicts in Python had no formally defined ordering. For Python version 3.6 and later, dicts are ordered by insertion order, see PEP 468. pandas will use the dict’s insertion order, when creating a Series or DataFrame from a dict and you’re using Python version 3.6 or higher. (GH19884)

Previous behavior (and current behavior if on Python < 3.6):

```python
In [16]: pd.Series({'Income': 2000,
                 ....:     'Expenses': -1500,
                 ....:     'Taxes': -200,
                 ....:     'Net result': 300})
Out[16]:
Expenses             -1500
Income                2000
Taxes                 -200
Net result           300
Length: 4, dtype: int64
```

Note the Series above is ordered alphabetically by the index values.

New behavior (for Python >= 3.6):

```python
In [74]: pd.Series({'Income': 2000,
                  ....:     'Expenses': -1500,
                  ....:     'Taxes': -200,
                  ....:     'Net result': 300})
Out[74]:
Income             2000
Expenses          -1500
Taxes             -200
Net result         300
Length: 4, dtype: int64
```

Notice that the Series is now ordered by insertion order. This new behavior is used for all relevant pandas types (Series, DataFrame, SparseSeries and SparseDataFrame).

If you wish to retain the old behavior while using Python >= 3.6, you can use .sort_index():
In [75]: pd.Series({'Income': 2000, 'Expenses': -1500, 'Taxes': -200, 'Net result': 300}).sort_index()

Out[75]:
Expenses    -1500
Income       2000
Net result   300
Taxes        -200
Length: 4, dtype: int64

**Deprecate Panel**

Panel was deprecated in the 0.20.x release, showing as a `DeprecationWarning`. Using Panel will now show a `FutureWarning`. The recommended way to represent 3-D data are with a MultiIndex on a DataFrame via the `to_frame()` or with the xarray package. pandas provides a `to_xarray()` method to automate this conversion (GH13563, GH18324).

In [75]: import pandas._testing as tm

In [76]: p = tm.makePanel()

In [77]: p

Out[77]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

Convert to a MultiIndex DataFrame

In [78]: p.to_frame()

Out[78]:

<table>
<thead>
<tr>
<th>major</th>
<th>minor</th>
<th>ItemA</th>
<th>ItemB</th>
<th>ItemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>A</td>
<td>0.469112</td>
<td>0.721555</td>
<td>0.404705</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-1.135632</td>
<td>0.271860</td>
<td>-1.039268</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.119209</td>
<td>0.276232</td>
<td>-1.344312</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-2.104569</td>
<td>0.113648</td>
<td>-0.109050</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>A</td>
<td>-0.282863</td>
<td>-0.706771</td>
<td>0.577046</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.212112</td>
<td>-0.424972</td>
<td>-0.370647</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-1.044236</td>
<td>-1.087401</td>
<td>0.844885</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>-0.494929</td>
<td>1.478427</td>
<td>1.643563</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>A</td>
<td>-1.509059</td>
<td>-1.039575</td>
<td>-1.715002</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>-0.173215</td>
<td>0.567020</td>
<td>-1.157892</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>-0.861849</td>
<td>-0.673690</td>
<td>1.075770</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>1.071804</td>
<td>0.524988</td>
<td>-1.469388</td>
</tr>
</tbody>
</table>

[12 rows x 3 columns]
Convert to an xarray DataArray

```python
In [79]: p.to_xarray()
Out[79]:
xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)
array([[ 0.469112, -1.135632,  0.119209, -2.104569],
       [-0.282863,  1.212112, -1.044236, -0.494929],
       [-1.509059, -0.173215, -0.861849,  1.071804]],
[[ 0.721555,  0.27186 ,  0.276232,  0.113648],
       [-0.706771, -0.424972, -1.087401, -1.478427],
       [-1.039575,  0.56702 , -0.67369 ,  0.524988]],
[[ 0.404705, -1.039268, -1.344312, -0.10905 ],
       [ 0.577046, -0.370647,  0.844885,  1.643563],
       [-1.715002, -1.157892,  1.07577 , -1.469388]])
Coordinates:
* items (items) object 'ItemA' 'ItemB' 'ItemC'
* major_axis (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05
* minor_axis (minor_axis) object 'A' 'B' 'C' 'D'
```

**pandas.core.common removals**

The following error & warning messages are removed from `pandas.core.common` (GH13634, GH19769):

- `PerformanceWarning`
- `UnsupportedFunctionCall`
- `UnsortedIndexError`
- `AbstractMethodError`

These are available from import from `pandas.errors` (since 0.19.0).

**Changes to make output of DataFrame.apply consistent**

`DataFrame.apply()` was inconsistent when applying an arbitrary user-defined-function that returned a list-like with `axis=1`. Several bugs and inconsistencies are resolved. If the applied function returns a Series, then pandas will return a DataFrame; otherwise a Series will be returned, this includes the case where a list-like (e.g. tuple or list is returned) (GH16353, GH17437, GH17970, GH17348, GH17892, GH18573, GH17602, GH18775, GH18901, GH18919).

```python
In [76]: df = pd.DataFrame(np.tile(np.arange(3), 6).reshape(6, -1) + 1,
                   columns=['A', 'B', 'C'])

In [77]: df
Out[77]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
(continues on next page)```
Previous behavior: if the returned shape happened to match the length of original columns, this would return a DataFrame. If the return shape did not match, a Series with lists was returned.

In [3]: df.apply(lambda x: [1, 2, 3], axis=1)
Out[3]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3

In [4]: df.apply(lambda x: [1, 2], axis=1)
Out[4]:
   0  [1, 2]
   1  [1, 2]
   2  [1, 2]
   3  [1, 2]
   4  [1, 2]
   5  [1, 2]
dtype: object

New behavior: When the applied function returns a list-like, this will now always return a Series.

In [78]: df.apply(lambda x: [1, 2, 3], axis=1)
Out[78]:
   0  [1, 2, 3]
   1  [1, 2, 3]
   2  [1, 2, 3]
   3  [1, 2, 3]
   4  [1, 2, 3]
   5  [1, 2, 3]
Length: 6, dtype: object

In [79]: df.apply(lambda x: [1, 2], axis=1)
Out[79]:
   0  [1, 2]
   1  [1, 2]
   2  [1, 2]
   3  [1, 2]
   4  [1, 2]
   5  [1, 2]
Length: 6, dtype: object

To have expanded columns, you can use result_type='expand'

In [80]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='expand')
Out[80]:

(continues on next page)
To broadcast the result across the original columns (the old behaviour for list-likes of the correct length), you can use `result_type='broadcast'`. The shape must match the original columns.

```
In [81]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='broadcast')
```

```
Out[81]:
       A  B  C
0   1  2  3
1   1  2  3
2   1  2  3
3   1  2  3
4   1  2  3
5   1  2  3
[6 rows x 3 columns]
```

Returning a `Series` allows one to control the exact return structure and column names:

```
In [82]: df.apply(lambda x: pd.Series([1, 2, 3], index=['D', 'E', 'F']), axis=1)
```

```
Out[82]:
         D  E  F
0   1  2  3
1   1  2  3
2   1  2  3
3   1  2  3
4   1  2  3
5   1  2  3
[6 rows x 3 columns]
```

**Concatenation will no longer sort**

In a future version of pandas `pandas.concat()` will no longer sort the non-concatenation axis when it is not already aligned. The current behavior is the same as the previous (sorting), but now a warning is issued when `sort` is not specified and the non-concatenation axis is not aligned (GH4588).

```
In [83]: df1 = pd.DataFrame({'a': [1, 2], 'b': [1, 2]}, columns=['b', 'a'])

In [84]: df2 = pd.DataFrame({'a': [4, 5]})

In [85]: pd.concat([df1, df2])
```

```
Out[85]: (continues on next page)
```
To keep the previous behavior (sorting) and silence the warning, pass `sort=True`

```python
In [86]: pd.concat([df1, df2], sort=True)
Out[86]:
     a  b
0  1.0 1
1  2.0 2
0  NaN 4
1  NaN 5
```

To accept the future behavior (no sorting), pass `sort=False`

Note that this change also applies to `DataFrame.append()`, which has also received a `sort` keyword for controlling this behavior.

**Build changes**

- Building pandas for development now requires `cython >= 0.24` (GH18613)
- Building from source now explicitly requires `setuptools` in `setup.py` (GH18113)
- Updated conda recipe to be in compliance with conda-build 3.0+ (GH18002)

**Index division by zero fills correctly**

Division operations on `Index` and subclasses will now fill division of positive numbers by zero with `np.inf`, division of negative numbers by zero with `-np.inf` and `0 / 0` with `np.nan`. This matches existing `Series` behavior. (GH19322, GH19347)

Previous behavior:

```python
In [6]: index = pd.Int64Index([-1, 0, 1])

In [7]: index / 0
Out[7]: Int64Index([0, 0, 0], dtype='int64')
```

# Previous behavior yielded different results depending on the type of zero in the divisor

```python
In [8]: index / 0.0
Out[8]: Float64Index([-inf, nan, inf], dtype='float64')
```

```python
In [9]: index = pd.UInt64Index([0, 1])
```
In [10]: index / np.array([0, 0], dtype=np.uint64)
Out[10]: UInt64Index([0, 0], dtype='uint64')

In [11]: pd.RangeIndex(1, 5) / 0
ZeroDivisionError: integer division or modulo by zero

Current behavior:

In [12]: index = pd.Int64Index([-1, 0, 1])
# division by zero gives -infinity where negative,
# +infinity where positive, and NaN for 0 / 0
In [13]: index / 0

# The result of division by zero should not depend on
# whether the zero is int or float
In [14]: index / 0.0

In [15]: index = pd.UInt64Index([0, 1])
In [16]: index / np.array([0, 0], dtype=np.uint64)
In [17]: pd.RangeIndex(1, 5) / 0

Extraction of matching patterns from strings

By default, extracting matching patterns from strings with str.extract() used to return a Series if a single group
was being extracted (a DataFrame if more than one group was extracted). As of pandas 0.23.0 str.extract() always
returns a DataFrame, unless expand is set to False. Finally, None was an accepted value for the expand parameter
(which was equivalent to False), but now raises a ValueError. (GH11386)

Previous behavior:

In [1]: s = pd.Series(['number 10', '12 eggs'])
In [2]: extracted = s.str.extract(r'.*(\d\d).*')
In [3]: extracted
Out [3]:
   0  10
   1  12
dtype: object
In [4]: type(extracted)
Out [4]:
pandas.core.series.Series

New behavior:

In [87]: s = pd.Series(['number 10', '12 eggs'])
In [88]: extracted = s.str.extract(r'.*(\d\d).*')
In [89]: extracted
Out[89]:
     0  10
   1  12
[2 rows x 1 columns]
In [90]: type(extracted)
Out[90]: pandas.core.frame.DataFrame

To restore previous behavior, simply set expand to False:

In [91]: s = pd.Series(['number 10', '12 eggs'])
In [92]: extracted = s.str.extract(r'.*(\d\d).*', expand=False)
In [93]: extracted
Out[93]:
     0  10
   1  12
Length: 2, dtype: object
In [94]: type(extracted)
Out[94]: pandas.core.series.Series

### Default value for the ordered parameter of CategoricalDtype

The default value of the ordered parameter for CategoricalDtype has changed from False to None to allow updating of categories without impacting ordered. Behavior should remain consistent for downstream objects, such as Categorical (GH18790)

In previous versions, the default value for the ordered parameter was False. This could potentially lead to the ordered parameter unintentionally being changed from True to False when users attempt to update categories if ordered is not explicitly specified, as it would silently default to False. The new behavior for ordered=None is to retain the existing value of ordered.

New behavior:

In [2]: from pandas.api.types import CategoricalDtype
In [3]: cat = pd.Categorical(list('abcaba'), ordered=True, categories=list('cba'))
In [4]: cat
Out[4]:
[a, b, c, a, b, a]
Categories (3, object): [c < b < a]
In [5]: cdt = CategoricalDtype(categories=list('cbad'))
In [6]: cat.astype(cdt)
Out[6]:
(continues on next page)
Categories (4, object): [c < b < a < d]

Notice in the example above that the converted Categorical has retained ordered=True. Had the default value for ordered remained as False, the converted Categorical would have become unordered, despite ordered=False never being explicitly specified. To change the value of ordered, explicitly pass it to the new dtype, e.g. CategoricalDtype(categories=list('cbad'), ordered=False).

Note that the unintentional conversion of ordered discussed above did not arise in previous versions due to separate bugs that prevented astype from doing any type of category to category conversion (GH10696, GH18593). These bugs have been fixed in this release, and motivated changing the default value of ordered.

Better pretty-printing of DataFrames in a terminal

Previously, the default value for the maximum number of columns was pd.options.display.max_columns=20. This meant that relatively wide data frames would not fit within the terminal width, and pandas would introduce line breaks to display these 20 columns. This resulted in an output that was relatively difficult to read:

If Python runs in a terminal, the maximum number of columns is now determined automatically so that the printed data frame fits within the current terminal width (pd.options.display.max_columns=0) (GH17023). If Python runs as a Jupyter kernel (such as the Jupyter QtConsole or a Jupyter notebook, as well as in many IDEs), this value cannot be inferred automatically and is thus set to 20 as in previous versions. In a terminal, this results in a much nicer output:
Note that if you don’t like the new default, you can always set this option yourself. To revert to the old setting, you can run this line:

```python
pd.options.display.max_columns = 20
```

**Datetimelike API changes**

- The default Timedelta constructor now accepts an ISO 8601 Duration string as an argument (GH19040)
- Subtracting NaT from a `Series` with dtype='datetime64[ns]' returns a Series with dtype='timedelta64[ns]' instead of dtype='datetime64[ns]' (GH18808)
- Addition or subtraction of NaT from `TimedeltaIndex` will return `TimedeltaIndex` instead of `DatetimeIndex` (GH19124)
- `DatetimeIndex.shift()` and `TimedeltaIndex.shift()` will now raise `NullFrequencyError` (which subclasses `ValueError`, which was raised in older versions) when the index object frequency is None (GH19147)
- Addition and subtraction of NaN from a `Series` with dtype='timedelta64[ns]' will raise a `TypeError` instead of treating the NaN as NaT (GH19274)
- NaT division with `datetime.timedelta` will now return NaN instead of raising (GH17876)
- Operations between a `Series` with dtype 'datetime64[ns]' and a `PeriodIndex` will correctly raises `TypeError` (GH18850)
• Subtraction of `Series` with timezone-aware dtype='datetime64[ns]' with mismatched timezones will raise `TypeError` instead of `ValueError` (GH18817)

• `Timestamp` will no longer silently ignore unused or invalid tz or tzinfo keyword arguments (GH17690)

• `Timestamp` will no longer silently ignore invalid freq arguments (GH5168)

• `CacheableOffset` and `WeekDay` are no longer available in the `pandas.tseries.offsets` module (GH17830)

• `pandas.tseries.frequencies.get_freq_group()` and `pandas.tseries.frequencies.DAYS` are removed from the public API (GH18034)

• `Series.truncate()` and `DataFrame.truncate()` will raise a `ValueError` if the index is not sorted instead of an unhelpful `KeyError` (GH17935)

• `Series.first` and `DataFrame.first` will now raise a `TypeError` rather than `NotImplementedError` when index is not a `DatetimeIndex` (GH20725).

• `Series.last` and `DataFrame.last` will now raise a `TypeError` rather than `NotImplementedError` when index is not a `DatetimeIndex` (GH20725).

• Restricted DateOffset keyword arguments. Previously, DateOffset subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176, GH18226).

• `pandas.merge()` provides a more informative error message when trying to merge on timezone-aware and timezone-naive columns (GH15800)

• For `DatetimeIndex` and `TimedeltaIndex` with freq=None, addition or subtraction of integer-dtyped array or Index will raise `NullFrequencyError` instead of `TypeError` (GH19895)

• `Timestamp` constructor now accepts a nanosecond keyword or positional argument (GH18898)

• `DatetimeIndex` will now raise an `AttributeError` when the tz attribute is set after instantiation (GH3746)

• `DatetimeIndex` with a `pytz` timezone will now return a consistent `pytz` timezone (GH18595)

Other API changes

• `Series.astype()` and `Index.astype()` with an incompatible dtype will now raise a `TypeError` rather than a `ValueError` (GH18231)

• Series construction with an object dtyped tz-aware datetime and dtype=object specified, will now return an object dtyped Series, previously this would infer the datetime dtype (GH18231)

• A `Series` of dtype=category constructed from an empty dict will now have categories of dtype=object rather than dtype=float64, consistently with the case in which an empty list is passed (GH18515)

• All-NaN levels in a `MultiIndex` are now assigned float rather than object dtype, promoting consistency with `Index` (GH17929).

• Levels names of a `MultiIndex` (when not None) are now required to be unique: trying to create a `MultiIndex` with repeated names will raise a `ValueError` (GH18872)

• Both construction and renaming of `Index/MultiIndex` with non-hashable name/names will now raise `TypeError` (GH20527)

• `Index.map()` can now accept `Series` and dictionary input objects (GH12756, GH18482, GH18509).

• `Dataframe.unstack()` will now default to filling with `np.nan` for object columns. (GH12815)

• `IntervalIndex` constructor will raise if the closed parameter conflicts with how the input data is inferred to be closed (GH18421)
- Inserting missing values into indexes will work for all types of indexes and automatically insert the correct type of missing value (Na\text{N}, Na\text{T}, etc.) regardless of the type passed in (GH18295)
- When created with duplicate labels, MultiIndex now raises a ValueError. (GH17464)
- Series.fillna() now raises a TypeError instead of a ValueError when passed a list, tuple or DataFrame as a value (GH18293)
- pandas.DataFrame.merge() no longer casts a float column to object when merging on int and float columns (GH16572)
- pandas.merge() now raises a ValueError when trying to merge on incompatible data types (GH9780)
- The default NA value for UInt64Index has changed from 0 to Na\text{N}, which impacts methods that mask with NA, such as UInt64Index.where() (GH18398)
- Refactored setup.py to use find_packages instead of explicitly listing out all subpackages (GH18535)
- Rearranged the order of keyword arguments in read_excel() to align with read_csv() (GH16672)
- wide_to_long() previously kept numeric-like suffixes as object dtype. Now they are cast to numeric if possible (GH17627)
- In read_excel(), the comment argument is now exposed as a named parameter (GH18735)
- Rearranged the order of keyword arguments in read_excel() to align with read_csv() (GH16672)
- The options html.border and mode.use_inf_as_null were deprecated in prior versions, these will now show FutureWarning rather than a DeprecationWarning (GH19003)
- IntervalIndex and IntervalDtype no longer support categorical, object, and string subtypes (GH19016)
- IntervalDtype now returns True when compared against 'interval' regardless of subtype, and IntervalDtype.name now returns 'interval' regardless of subtype (GH18980)
- KeyError now raises instead of ValueError in drop(), drop(), drop(), drop() when dropping a non-existent element in an axis with duplicates (GH19186)
- Series.to_csv() now accepts a compression argument that works in the same way as the compression argument in DataFrame.to_csv() (GH18958)
- Set operations (union, difference...) on IntervalIndex with incompatible index types will now raise a TypeError rather than a ValueError (GH19329)
- DateOffset objects render more simply, e.g. <DateOffset: days=1> instead of <DateOffset: kwds={"days": 1}> (GH19403)
- Categorical.fillna now validates its value and method keyword arguments. It now raises when both or none are specified, matching the behavior of Series.fillna() (GH19682)
- pd.to_datetime('today') now returns a datetime, consistent with pd.Timestamp('today'): previously pd.to_datetime('today') returned a .normalized() datetime (GH19935)
- Series.str.replace() now takes an optional regex keyword which, when set to False, uses literal string replacement rather than regex replacement (GH16808)
- DatetimeIndex.strftime() and PeriodIndex.strftime() now return an Index instead of a numpy array to be consistent with similar accessors (GH20127)
- Constructing a Series from a list of length 1 no longer broadcasts this list when a longer index is specified (GH19714, GH20391)
- DataFrame.to_dict() with orient='index' no longer casts int columns to float for a DataFrame with only int and float columns (GH18580)
• A user-defined-function that is passed to `Series.rolling().aggregate()`, `DataFrame.rolling().aggregate()`, or its expanding cousins, will now always be passed a `Series`, rather than a `np.array`. `.apply()` only has the `raw` keyword, see here. This is consistent with the signatures of `.aggregate()` across pandas (GH20584)

• Rolling and Expanding types raise `NotImplementedError` upon iteration (GH11704).

### Deprecations

• `Series.from_array` and `SparseSeries.from_array` are deprecated. Use the normal constructor `Series(.)` and `SparseSeries(.)` instead (GH18213).

• `DataFrame.as_matrix` is deprecated. Use `DataFrame.values` instead (GH18458).

• `Series.astype`, `DatetimeIndex.astype`, `PeriodIndex.astype` and `TimeDeltaIndex.astype` have been deprecated. Use `.astype(object)` instead (GH18572)

• Grouping by a tuple of keys now emits a `FutureWarning` and is deprecated. In the future, a tuple passed to 'by' will always refer to a single key that is the actual tuple, instead of treating the tuple as multiple keys. To retain the previous behavior, use a list instead of a tuple (GH18314)

• `Series.valid` is deprecated. Use `Series.dropna()` instead (GH18800).

• `read_excel()` has deprecated the `skip_footer` parameter. Use `skipfooter` instead (GH18836)

• `ExcelFile.parse()` has deprecated `sheetname` in favor of `sheet_name` for consistency with `read_excel()` (GH20920).

• The `is_copy` attribute is deprecated and will be removed in a future version (GH18801).

• `IntervalIndex.from_intervals` is deprecated in favor of the `IntervalIndex` constructor (GH19263)

• `DataFrame.from_items` is deprecated. Use `DataFrame.from_dict()` instead, or `DataFrame.from_dict(OrderedDict())` if you wish to preserve the key order (GH17320, GH17312)

• Indexing a `MultiIndex` or a `FloatIndex` with a list containing some missing keys will now show a `FutureWarning`, which is consistent with other types of indexes (GH17758).

• The `broadcast` parameter of `.apply()` is deprecated in favor of `result_type='broadcast'` (GH18577)

• The `reduce` parameter of `.apply()` is deprecated in favor of `result_type='reduce'` (GH18577)

• The `order` parameter of `factorize()` is deprecated and will be removed in a future release (GH19727)

• `Timestamp.weekday_name`, `DatetimeIndex.weekday_name`, and `Series.dt.weekday_name` are deprecated in favor of `Timestamp.day_name()`, `DatetimeIndex.day_name()`, and `Series.dt.day_name()` (GH12806)

• `pandas.tseries.plotting.tsplot` is deprecated. Use `Series.plot()` instead (GH18627)

• `Index.summary()` is deprecated and will be removed in a future version (GH18217)

• `NDFrame.get_ftype_counts()` is deprecated and will be removed in a future version (GH18243)

• The `convert_datetime64` parameter in `DataFrame.to_records()` has been deprecated and will be removed in a future version. The NumPy bug motivating this parameter has been resolved. The default value for this parameter has also changed from `True` to `None` (GH18160).

• `Series.rolling().apply()`, `DataFrame.rolling().apply()`, `Series.expanding().apply()`, and `DataFrame.expanding().apply()` have deprecated passing an `np.array` by default. One will need to pass the new `raw` parameter to be explicit about what is passed (GH20584)

• The `data`, `base`, `strides`, `flags` and `itemsize` properties of the `Series` and `Index` classes have been deprecated and will be removed in a future version (GH20419).
• DatetimeIndex.offset is deprecated. Use DatetimeIndex.freq instead (GH20716)

• Floor division between an integer ndarray and a Timedelta is deprecated. Divide by Timedelta.value instead (GH19761)

• Setting PeriodIndex.freq (which was not guaranteed to work correctly) is deprecated. Use PeriodIndex.asfreq() instead (GH20678)

• Index.get_duplicates() is deprecated and will be removed in a future version (GH20239)

• The previous default behavior of negative indices in Categorical.take is deprecated. In a future version it will change from meaning missing values to meaning positional indices from the right. The future behavior is consistent with Series.take() (GH20664).

• Passing multiple axes to the axis parameter in DataFrame.dropna() has been deprecated and will be removed in a future version (GH20987)

Removal of prior version deprecations/changes

• Warnings against the obsolete usage Categorical(codes, categories), which were emitted for instance when the first two arguments to Categorical() had different dtypes, and recommended the use of Categorical.from_codes, have now been removed (GH8074)

• The levels and labels attributes of a MultiIndex can no longer be set directly (GH4039).

• pd.tseries.util.pivot_annual has been removed (deprecated since v0.19). Use pivot_table instead (GH18370)

• pd.tseries.util.isleapyear has been removed (deprecated since v0.19). Use .is_leap_year property in Datetime-like instead (GH18370)

• pd.ordered_merge has been removed (deprecated since v0.19). Use pd.merge_ordered instead (GH18459)

• The SparseList class has been removed (GH14007)

• The pandas.io.wb and pandas.io.data stub modules have been removed (GH13735)

• Categorical.from_array has been removed (GH13854)

• The freq and how parameters have been removed from the rolling/expanding/ewm methods of DataFrame and Series (deprecated since v0.18). Instead, resample before calling the methods. (GH18601 & GH18668)

• DatetimeIndex.to_datetime, Timestamp.to_datetime, PeriodIndex.to_datetime, and Index.to_datetime have been removed (GH8254, GH14096, GH14113)

• read_csv() has dropped the skip_footer parameter (GH13386)

• read_csv() has dropped the as_recarray parameter (GH13373)

• read_csv() has dropped the buffer_lines parameter (GH13360)

• read_csv() has dropped the compact_ints and use_unsigned parameters (GH13323)

• The Timestamp class has dropped the offset attribute in favor of freq (GH13593)

• The Series, Categorical, and Index classes have dropped the reshape method (GH13012)

• pandas.tseries.frequencies.get_standard_freq has been removed in favor of pandas.tseries.frequencies.to_offset(freq).rule_code (GH13874)

• The freqstr keyword has been removed from pandas.tseries.frequencies.to_offset in favor of freq (GH13874)

• The Panel4D and PanelND classes have been removed (GH13776)
• The Panel class has dropped the to_long and toLong methods (GH19077)
• The options display.line_with and display.height are removed in favor of display.width and display.max_rows respectively (GH4391, GH19107)
• The labels attribute of the Categorical class has been removed in favor of Categorical.codes (GH7768)
• The flavor parameter have been removed from func:to_sql method (GH13611)
• The modules pandas.tools.hashing and pandas.util.hashing have been removed (GH16223)
• The top-level functions pd.rolling_* , pd.expanding_* and pd.ewm* have been removed (Deprecated since v0.18). Instead, use the DataFrame/Series methods rolling, expanding and ewm (GH18723)
• Imports from pandas.core.common for functions such as is_datetime64_dtype are now removed. These are located in pandas.api.types. (GH13634, GH19769)
• The infer_dst keyword in Series.tz_localize(), DatetimeIndex.tz_localize() and DatetimeIndex have been removed. infer_dst=True is equivalent to ambiguous='infer', and infer_dst=False to ambiguous='raise' (GH7963).
• When .resample() was changed from an eager to a lazy operation, like .groupby() in v0.18.0, we put in place compatibility (with a FutureWarning), so operations would continue to work. This is now fully removed, so a Resampler will no longer forward compat operations (GH20554)
• Remove long deprecated axis=None parameter from .replace() (GH20271)

Performance improvements

• Indexers on Series or DataFrame no longer create a reference cycle (GH17956)
• Added a keyword argument, cache, to to_datetime() that improved the performance of converting duplicate datetime arguments (GH11665)
• DateOffset arithmetic performance is improved (GH18218)
• Converting a Series of Timedelta objects to days, seconds, etc... sped up through vectorization of underlying methods (GH18092)
• Improved performance of .map() with a Series/dict input (GH15081)
• The overridden Timedelta properties of days, seconds and microseconds have been removed, leveraging their built-in Python versions instead (GH18242)
• Series construction will reduce the number of copies made of the input data in certain cases (GH17449)
• Improved performance of Series.dt.date() and DatetimeIndex.date() (GH18058)
• Improved performance of Series.dt.time() and DatetimeIndex.time() (GH18461)
• Improved performance of IntervalIndex.symmetric_difference() (GH18475)
• Improved performance of DatetimeIndex and Series arithmetic operations with Business-Month and Business-Quarter frequencies (GH18489)
• Series() / DataFrame() tab completion limits to 100 values, for better performance. (GH18587)
• Improved performance of DataFrame.median() with axis=1 when bottleneck is not installed (GH16468)
• Improved performance of MultiIndex.get_loc() for large indexes, at the cost of a reduction in performance for small ones (GH18519)
• Improved performance of MultiIndex.remove_unused_levels() when there are no unused levels, at the cost of a reduction in performance when there are (GH19289)
• Improved performance of `Index.get_loc()` for non-unique indexes (GH19478)
• Improved performance of pairwise `.rolling()` and `.expanding()` with `.cov()` and `.corr()` operations (GH17917)
• Improved performance of `pandas.core.groupby.GroupBy.rank()` (GH15779)
• Improved performance of variable `.rolling()` on `.min()` and `.max()` (GH19521)
• Improved performance of `pandas.core.groupby.GroupBy.ffill()` and `pandas.core.groupby.GroupBy.bfill()` (GH11296)
• Improved performance of `pandas.core.groupby.GroupBy.any()` and `pandas.core.groupby.GroupBy.all()` (GH15435)
• Improved performance of `pandas.core.groupby.GroupBy.pct_change()` (GH19165)
• Improved performance of `Series.isin()` in the case of categorical dtypes (GH20003)
• Improved performance of `getattr(Series, attr)` when the Series has certain index types. This manifested in slow printing of large Series with a `DatetimeIndex` (GH19764)
• Fixed a performance regression for `GroupBy.nth()` and `GroupBy.last()` with some object columns (GH19283)
• Improved performance of `pandas.core.arrays.Categorical.from_codes()` (GH18501)

Documentation changes

Thanks to all of the contributors who participated in the pandas Documentation Sprint, which took place on March 10th. We had about 500 participants from over 30 locations across the world. You should notice that many of the API docstrings have greatly improved.

There were too many simultaneous contributions to include a release note for each improvement, but this GitHub search should give you an idea of how many docstrings were improved.

Special thanks to Marc Garcia for organizing the sprint. For more information, read the NumFOCUS blogpost recapping the sprint.

• Changed spelling of “numpy” to “NumPy”, and “python” to “Python”. (GH19017)
• Consistency when introducing code samples, using either colon or period. Rewrote some sentences for greater clarity, added more dynamic references to functions, methods and classes. (GH18941, GH18948, GH18973, GH19017)
• Added a reference to `DataFrame.assign()` in the concatenate section of the merging documentation (GH18665)

Bug fixes

Categorical

Warning: A class of bugs were introduced in pandas 0.21 with `CategoricalDtype` that affects the correctness of operations like `merge`, `concat`, and indexing when comparing multiple unordered `Categorical` arrays that have the same categories, but in a different order. We highly recommend upgrading or manually aligning your categories before doing these operations.
Bug in `Categorical.equals` returning the wrong result when comparing two unordered `Categorical` arrays with the same categories, but in a different order (GH16603)

Bug in `pandas.api.types.union_categoricals()` returning the wrong result when for unordered categoricals with the categories in a different order. This affected `pandas.concat()` with Categorical data (GH19096).

Bug in `pandas.merge()` returning the wrong result when joining on an unordered Categorical that had the same categories but in a different order (GH19551)

Bug in `CategoricalIndex.get_indexer()` returning the wrong result when `target` was an unordered `Categorical` that had the same categories as `self` but in a different order (GH19551)

Bug in `Index.astype()` with a categorical dtype where the resultant index is not converted to a `CategoricalIndex` for all types of index (GH18630)

Bug in `Series.astype()` and `Categorical.astype()` where an existing categorical data does not get updated (GH10696, GH18593)

Bug in `Series.str.split()` with `expand=True` incorrectly raising an `IndexError` on empty strings (GH20002).

Bug in `Index` constructor with `dtype=CategoricalDtype(...)` where categories and `ordered` are not maintained (GH19032)

Bug in `Series` constructor with scalar and `dtype=CategoricalDtype(...)` where categories and `ordered` are not maintained (GH19565)

Bug in `Categorical.__iter__` not converting to Python types (GH19909)

Bug in `pandas.factorize()` returning the unique codes for the `uniques` This now returns a `Categorical` with the same dtype as the input (GH19721)

Bug in `pandas.factorize()` including an item for missing values in the `uniques` return value (GH19721)

Bug in `Series.take()` with categorical data interpreting -1 in `indices` as missing value markers, rather than the last element of the Series (GH20664)

**Datetimelike**

Bug in `Series.__sub__()` subtracting a non-nanosecond `np.datetime64` object from a `Series` gave incorrect results (GH7996)

Bug in `DatetimeIndex`, `TimedeltaIndex` addition and subtraction of zero-dimensional integer arrays gave incorrect results (GH19012)

Bug in `DatetimeIndex` and `TimedeltaIndex` where adding or subtracting an array-like of `DateOffset` objects either raised (`np.array`, `pd.Index`) or broadcast incorrectly (`pd.Series`) (GH18849)

Bug in `Series.__add__()` adding `Series` with `dtype=timedelta64[ns]` to a timezone-aware `DatetimeIndex` incorrectly dropped timezone information (GH13905)

Adding a `Period` object to a `datetime` or `Timestamp` object will now correctly raise a `TypeError` (GH17983)

Bug in `Timestamp` where comparison with an array of `Timestamp` objects would result in a `RecursionError` (GH15183)

Bug in `Series` floor-division where operating on a scalar `timedelta` raises an exception (GH18846)

Bug in `DatetimeIndex` where the repr was not showing high-precision time values at the end of a day (e.g., 23:59:59.999999999) (GH19030)

Bug in `.astype()` to non-ns `timedelta` units would hold the incorrect `dtype` (GH19176, GH19223, GH12425)
• Bug in subtracting `Series` from `NaT` incorrectly returning `NaT` (GH19158)
• Bug in `Series.truncate()` which raises `TypeError` with a monotonic `PeriodIndex` (GH17717)
• Bug in `pct_change()` using periods and `freq` returned different length outputs (GH7292)
• Bug in comparison of `DatetimeIndex` against `None` or `datetime.date` objects raising `TypeError` for `==` and `!=` comparisons instead of all-`False` and all-`True`, respectively (GH19301)
• Bug in `Timestamp` and `to_datetime()` where a string representing a barely out-of-bounds timestamp would be incorrectly rounded down instead of raising `OutOfBoundsDatetime` (GH19382)
• Bug in `to_datetime()` where passing an out-of-bounds datetime with `errors='coerce'` and `utc=True` would raise `OutOfBoundsDatetime` instead of parsing to `NaT` (GH19612)
• Bug in `DatetimeIndex` and `TimedeltaIndex` addition and subtraction where name of the returned object was not always set consistently. (GH19744)
• Bug in `DatetimedeltaIndex` addition and subtraction where operations with numpy arrays raised `TypeError` (GH19847)
• Bug in `DatetimedeltaIndex` where setting the `freq` attribute was not fully supported (GH20678)

**Timedelta**

• Bug in `Timedelta.__mul__()` where multiplying by `NaT` returned `NaT` instead of raising a `TypeError` (GH19819)
• Bug in `Series` with `dtype='timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` had results cast to `dtype='int64'` (GH17250)
• Bug in `Series` with `dtype='timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` could return a `Series` with an incorrect name (GH19043)
• Bug in `Timedelta.__floordiv__()` and `Timedelta.__rfloordiv__()` dividing by many incompatible numpy objects was incorrectly allowed (GH18846)
• Bug where dividing a scalar timedelta-like object with `TimedeltaIndex` performed the reciprocal operation (GH19125)
• Bug in `TimedeltaIndex` where division by a `Series` would return a `TimedeltaIndex` instead of a `Series` (GH19042)
• Bug in `Timedelta.__add__()`, `Timedelta.__sub__()` where adding or subtracting a `np.timedelta64` object would return another `np.timedelta64` instead of a `Timedelta` (GH19738)
• Bug in `Timedelta.__floordiv__()`, `Timedelta.__rfloordiv__()` where operating with a `Tick` object would raise a `TypeError` instead of returning a numeric value (GH19738)
• Bug in `Period.asfreq()` where periods near `datetime(1, 1, 1)` could be converted incorrectly (GH19643, GH19834)
• Bug in `Timedelta.total_seconds()` causing precision errors, for example `Timedelta('30S').total_seconds()==30.000000000000004` (GH19458)
• Bug in `Timedelta.__rmod__()` where operating with a `numpy.timedelta64` returned a `timedelta64` object instead of a `Timedelta` (GH19820)
Multiplication of `TimedeltaIndex` by `TimedeltaIndex` will now raise `TypeError` instead of raising `ValueError` in cases of length mismatch (GH19333)

Bug in indexing a `TimedeltaIndex` with a `np.timedelta64` object which was raising a `TypeError` (GH20393)

**Timezones**

- Bug in creating a `Series` from an array that contains both tz-naive and tz-aware values will result in a `Series` whose dtype is tz-aware instead of object (GH16406)
- Bug in comparison of timezone-aware `DatetimeIndex` against NaT incorrectly raising `TypeError` (GH19276)
- Bug in `Datetimex.pastype()` when converting between timezone aware dtypes, and converting from timezone aware to naive (GH18951)
- Bug in comparing `Datetimex` which failed to raise `TypeError` when attempting to compare timezone-aware and timezone-naive datetimelike objects (GH18162)
- Bug in localization of a naive, datetime string in a `Series` constructor with a `datetime64[ns, tz]` dtype (GH174151)
- `Timestamp.replace()` will now handle Daylight Savings transitions gracefully (GH18319)
- Bug in tz-aware `Datetimex` where addition/subtraction with a `TimedeltaIndex` or array with dtype='timedelta64[ns]' was incorrect (GH17558)
- Bug in `Datetimex.insert()` where inserting NaT into a timezone-aware index incorrectly raised (GH16357)
- Bug in `DataFrame` constructor, where tz-aware Datetimeindex and a given column name will result in an empty `DataFrame` (GH19157)
- Bug in `Timestamp.tz_localize()` where localizing a timestamp near the minimum or maximum valid values could overflow and return a timestamp with an incorrect nanosecond value (GH12677)
- Bug when iterating over `Datetimex` that was localized with fixed timezone offset that rounded nanosecond precision to microseconds (GH19603)
- Bug in `DataFrame.diff()` that raised an `IndexError` with tz-aware values (GH18578)
- Bug in `melt()` that converted tz-aware dtypes to tz-naive (GH15785)
- Bug in `Dataframe.count()` that raised an `ValueError`, if `Dataframe.dropna()` was called for a single column with timezone-aware values. (GH13407)

**Offsets**

- Bug in `WeekOfMonth` and `Week` where addition and subtraction did not roll correctly (GH18510, GH18672, GH18864)
- Bug in `WeekOfMonth` and `LastWeekOfMonth` where default keyword arguments for constructor raised `ValueError` (GH19142)
- Bug in `FY5253Quarter, LastWeekOfMonth` where rollback and rollforward behavior was inconsistent with addition and subtraction behavior (GH18854)
- Bug in `FY5253` where datetime addition and subtraction incremented incorrectly for dates on the year-end but not normalized to midnight (GH18854)
• Bug in FY5253 where date offsets could incorrectly raise an `AssertionError` in arithmetic operations (GH14774)

**Numeric**

• Bug in `Series` constructor with an int or float list where specifying `dtype=str`, `dtype='str'` or `dtype='U'` failed to convert the data elements to strings (GH16605)
• Bug in `Index` multiplication and division methods where operating with a `Series` would return an `Index` object instead of a `Series` object (GH19042)
• Bug in the `DataFrame` constructor in which data containing very large positive or very large negative numbers was causing `OverflowError` (GH18584)
• Bug in `Index` constructor with `dtype='uint64'` where int-like floats were not coerced to `UInt64Index` (GH18400)
• Bug in `DataFrame` flex arithmetic (e.g. `df.add(other, fill_value=foo)`) with a `fill_value` other than `None` failed to raise `NotImplementedError` in corner cases where either the frame or `other` has length zero (GH19522)
• Multiplication and division of numeric-dtyped `Index` objects with timedelta-like scalars returns `TimedeltaIndex` instead of raising `TypeError` (GH19333)
• Bug where `NaN` was returned instead of 0 by `Series.pct_change()` and `DataFrame.pct_change()` when `fill_method` is not `None` (GH19873)

**Strings**

• Bug in `Series.str.get()` with a dictionary in the values and the index not in the keys, raising `KeyError` (GH20671)

**Indexing**

• Bug in `Index` construction from list of mixed type tuples (GH18505)
• Bug in `Index.drop()` when passing a list of both tuples and non-tuples (GH18304)
• Bug in `DataFrame.drop()`, `Panel.drop()`, `Series.drop()`, `Index.drop()` where no `KeyError` is raised when dropping a non-existent element from an axis that contains duplicates (GH19186)
• Bug in indexing a datetimelike `Index` that raised `ValueError` instead of `IndexError` (GH18386).
• `Index.to_series()` now accepts `index` and `name` kwargs (GH18699)
• `DatetimeIndex.to_series()` now accepts `index` and `name` kwargs (GH18699)
• Bug in indexing non-scalar value from `Series` having non-unique `Index` will return value flattened (GH17610)
• Bug in indexing with iterator containing only missing keys, which raised no error (GH20748)
• Fixed inconsistency in `.ix` between list and scalar keys when the index has integer dtype and does not include the desired keys (GH20753)
• Bug in `__setitem__` when indexing a `DataFrame` with a 2-d boolean ndarray (GH18582)
• Bug in `str.extractall` when there were no matches empty `Index` was returned instead of appropriate `MultiIndex` (GH19034)
• Bug in `IntervalIndex` where empty and purely NA data was constructed inconsistently depending on the construction method (GH18421)
• Bug in `IntervalIndex.symmetric_difference()` where the symmetric difference with a non-`IntervalIndex` did not raise (GH18475)
• Bug in `IntervalIndex` where set operations that returned an empty `IntervalIndex` had the wrong dtype (GH19101)
• Bug in `DataFrame.drop_duplicates()` where no `KeyError` is raised when passing in columns that don’t exist on the `DataFrame` (GH19726)
• Bug in `Index` subclasses constructors that ignore unexpected keyword arguments (GH19348)
• Bug in `Index.difference()` when taking difference of an `Index` with itself (GH20040)
• Bug in `DataFrame.first_valid_index()` and `DataFrame.last_valid_index()` in presence of entire rows of NaNs in the middle of values (GH20499).
• Bug in `IntervalIndex` where some indexing operations were not supported for overlapping or non-monotonic `uint64` data (GH20636)
• Bug in `Series.is_unique` where extraneous output in stderr is shown if Series contains objects with `__ne__` defined (GH20661)
• Bug in `.loc` assignment with a single-element list-like incorrectly assigns as a list (GH19474)
• Bug in partial string indexing on a `Series/DataFrame` with a monotonic decreasing `DatetimeIndex` (GH19362)
• Bug in performing in-place operations on a `DataFrame` with a duplicate `Index` (GH17105)
• Bug in `IntervalIndex.get_loc()` and `IntervalIndex.get_indexer()` when used with an `IntervalIndex` containing a single interval (GH17284, GH20921)
• Bug in `.loc` with a `uint64` indexer (GH20722)

`MultiIndex`

• Bug in `MultiIndex.__contains__()` where non-tuple keys would return True even if they had been dropped (GH19027)
• Bug in `MultiIndex.set_labels()` which would cause casting (and potentially clipping) of the new labels if the `level` argument is not 0 or a list like `[0, 1, ... ]` (GH19057)
• Bug in `MultiIndex.get_level_values()` which would return an invalid index on level of ints with missing values (GH17924)
• Bug in `MultiIndex.unique()` when called on empty `MultiIndex` (GH20568)
• Bug in `MultiIndex.unique()` which would not preserve level names (GH20570)
• Bug in `MultiIndex.remove_unused_levels()` which would fill nan values (GH18417)
• Bug in `MultiIndex.from_tuples()` which would fail to take zipped tuples in python3 (GH18434)
• Bug in `MultiIndex.get_loc()` which would fail to automatically cast values between float and int (GH18818, GH15994)
• Bug in `MultiIndex.get_loc()` which would cast boolean to integer labels (GH19086)
• Bug in `MultiIndex.get_loc()` which would fail to locate keys containing NaN (GH18485)
• Bug in `MultiIndex.get_loc()` in large `MultiIndex`, would fail when levels had different dtypes (GH18520)
• Bug in indexing where nested indexers having only numpy arrays are handled incorrectly (GH19686)

IO

• `read_html()` now rewrites seekable IO objects after parse failure, before attempting to parse with a new parser. If a parser errors and the object is non-seekable, an informative error is raised suggesting the use of a different parser (GH17975)

• `DataFrame.to_html()` now has an option to add an id to the leading `<table>` tag (GH8496)

• Bug in `read_msgpack()` with a non-existent file is passed in Python 2 (GH15296)

• Bug in `read_csv()` where a MultiIndex with duplicate columns was not being mangled appropriately (GH18062)

• Bug in `read_csv()` where missing values were not being handled properly when `keep_default_na=False` with dictionary `na_values` (GH19227)

• Bug in `read_csv()` causing heap corruption on 32-bit, big-endian architectures (GH20785)

• Bug in `read_sas()` where a file with 0 variables gave an AttributeError incorrectly. Now it gives an EmptyDataError (GH18184)

• Bug in `DataFrame.to_latex()` where pairs of braces meant to serve as invisible placeholders were escaped (GH18667)

• Bug in `DataFrame.to_latex()` where a NaN in a MultiIndex would cause an IndexError or incorrect output (GH14249)

• Bug in `DataFrame.to_latex()` where a non-string index-level name would result in an AttributeError (GH19981)

• Bug in `DataFrame.to_latex()` where the combination of an index name and the `index_names=False` option would result in incorrect output (GH18326)

• Bug in `DataFrame.to_latex()` where a MultiIndex with an empty string as its name would result in incorrect output (GH18669)

• Bug in `DataFrame.to_latex()` where missing space characters caused wrong escaping and produced non-valid latex in some cases (GH20859)

• Bug in `read_json()` where large numeric values were causing an OverflowError (GH18842)

• Bug in `DataFrame.to_parquet()` where an exception was raised if the write destination is S3 (GH19134)

• `Interval` now supported in `DataFrame.to_excel()` for all Excel file types (GH19242)

• `Timedelta` now supported in `DataFrame.to_excel()` for all Excel file types (GH19242, GH9155, GH19900)

• Bug in `pandas.io.stata.StataReader.value_labels()` raising an AttributeError when called on very old files. Now returns an empty dict (GH19417)

• Bug in `read_pickle()` when unpickling objects with `TimedeltaIndex` or `Float64Index` created with pandas prior to version 0.20 (GH19939)

• Bug in `pandas.io.json.json_normalize()` where sub-records are not properly normalized if any sub-records values are NoneType (GH20030)

• Bug in `usecols` parameter in `read_csv()` where error is not raised correctly when passing a string. (GH20529)

• Bug in `HDFStore.keys()` when reading a file with a soft link causes exception (GH20523)

• Bug in `HDFStore.select_column()` where a key which is not a valid store raised an AttributeError instead of a KeyError (GH17912)
Plotting

- Better error message when attempting to plot but matplotlib is not installed (GH19810).
- DataFrame.plot() now raises a `ValueError` when the x or y argument is improperly formed (GH18671)
- Bug in DataFrame.plot() when x and y arguments given as positions caused incorrect referenced columns for line, bar and area plots (GH20056)
- Bug in formatting tick labels with `datetime.time()` and fractional seconds (GH18478).
- Series.plot.kde() has exposed the args `ind` and `bw_method` in the docstring (GH18461). The argument `ind` may now also be an integer (number of sample points).
- DataFrame.plot() now supports multiple columns to the y argument (GH19699)

GroupBy/resample/rolling

- Bug when grouping by a single column and aggregating with a class like `list` or `tuple` (GH18079)
- Fixed regression in DataFrame.groupby() which would not emit an error when called with a tuple key not in the index (GH18798)
- Bug in DataFrame.resample() which silently ignored unsupported (or mistyped) options for `label`, `closed` and `convention` (GH19303)
- Bug in DataFrame.groupby() where tuples were interpreted as lists of keys rather than as keys (GH17979, GH18249)
- Bug in DataFrame.groupby() where aggregation by `first/last/min/max` was causing timestamps to lose precision (GH19526)
- Bug in DataFrame.transform() where particular aggregation functions were being incorrectly cast to match the dtype(s) of the grouped data (GH19200)
- Bug in DataFrame.groupby() passing the `on` kwarg, and subsequently using `.apply()` (GH17813)
- Bug in DataFrame.resample().aggregate not raising a `KeyError` when aggregating a non-existent column (GH16766, GH19566)
- Bug in DataFrameGroupBy.cumsum() and DataFrameGroupBy.cumprod() when `skipna` was passed (GH19806)
- Bug in DataFrame.resample() that dropped timezone information (GH13238)
- Bug in DataFrame.groupby() where transformations using `np.all` and `np.any` were raising a `ValueError` (GH20653)
- Bug in DataFrame.resample() where `ffill`, `bfill`, `pad`, `backfill`, `fillna`, `interpolate`, and `asfreq` were ignoring `loffset`. (GH20744)
- Bug in DataFrame.groupby() when applying a function that has mixed data types and the user supplied function can fail on the grouping column (GH20949)
- Bug in DataFrameGroupBy.rolling().apply() where operations performed against the associated DataFrameGroupBy object could impact the inclusion of the grouped item(s) in the result (GH14013)
Sparse

- Bug in which creating a SparseDataFrame from a dense Series or an unsupported type raised an uncontrolled exception (GH19374)
- Bug in SparseDataFrame.to_csv causing exception (GH19384)
- Bug in SparseSeries.memory_usage which caused segfault by accessing non sparse elements (GH19368)
- Bug in constructing a SparseArray: if data is a scalar and index is defined it will coerce to float64 regardless of scalar's dtype. (GH19163)

Reshaping

- Bug in DataFrame.merge() where referencing a CategoricalIndex by name, where the by kwarg would KeyError (GH20777)
- Bug in DataFrame.stack() which fails trying to sort mixed type levels under Python 3 (GH18310)
- Bug in DataFrame.unstack() which casts int to float if columns is a MultiIndex with unused levels (GH17845)
- Bug in DataFrame.unstack() which raises an error if index is a MultiIndex with unused labels on the unstacked level (GH18562)
- Fixed construction of a Series from a dict containing NaN as key (GH18480)
- Fixed construction of a DataFrame from a dict containing NaN as key (GH18455)
- Disabled construction of a Series where len(index) > len(data) = 1, which previously would broadcast the data item, and now raises a ValueError (GH18819)
- Suppressed error in the construction of a DataFrame from a dict containing scalar values when the corresponding keys are not included in the passed index (GH18600)
- Fixed (changed from object to float64) dtype of DataFrame initialized with axes, no data, and dtype=int (GH19646)
- Bug in Series.rank() where Series containing NaT modifies the Series inplace (GH18521)
- Bug in cut() which fails when using readonly arrays (GH18773)
- Bug in DataFrame.pivot_table() which fails when the aggfunc arg is of type string. The behavior is now consistent with other methods like agg and apply (GH18713)
- Bug in DataFrame.merge() in which merging using Index objects as vectors raised an Exception (GH19038)
- Bug in DataFrame.stack(), DataFrame.unstack(), Series.unstack() which were not returning sub-classes (GH15563)
- Bug in timezone comparisons, manifesting as a conversion of the index to UTC in .concat() (GH18523)
- Bug in concat() when concatenating sparse and dense series it returns only a SparseDataFrame. Should be a DataFrame. (GH18914, GH18686, and GH16874)
- Improved error message for DataFrame.merge() when there is no common merge key (GH19427)
- Bug in DataFrame.join() which does an outer instead of a left join when being called with multiple DataFrames and some have non-unique indices (GH19624)
- Series.rename() now accepts axis as a kwarg (GH18589)
- Bug in rename() where an Index of same-length tuples was converted to a MultiIndex (GH19497)
• Comparisons between *Series* and *Index* would return a *Series* with an incorrect name, ignoring the *Index*’s name attribute (GH19582)

• Bug in *qcut()* where datetime and timedelta data with NaT present raised a *ValueError* (GH19768)

• Bug in *DataFrame.iterrows()*, which would infer strings not compliant to ISO8601 to datetimes (GH19671)

• Bug in *Series* constructor with *Categorical* where a *ValueError* is not raised when an index of different length is given (GH19342)

• Bug in *DataFrame.astype()* where column metadata is lost when converting to categorical or a dictionary of dtypes (GH19920)

• Bug in *cut()* and *qcut()* where timezone information was dropped (GH19872)

• Bug in *Series* constructor with a *dtype=str*, previously raised in some cases (GH19853)

• Bug in *get_dummies()* and *select_dtypes()* where duplicate column names caused incorrect behavior (GH20848)

• Bug in *isna()* which cannot handle ambiguous typed lists (GH20675)

• Bug in *concat()* which raises an error when concatenating TZ-aware dataframes and all-NaT dataframes (GH12396)

• Bug in *concat()* which raises an error when concatenating empty TZ-aware series (GH18447)

**Other**

• Improved error message when attempting to use a Python keyword as an identifier in a *numexpr* backed query (GH18221)

• Bug in accessing a *pandas.get_option()* which raised *KeyError* rather than *OptionError* when looking up a non-existent option key in some cases (GH19789)

• Bug in *testing.assert_series_equal()* and *testing.assert_frame_equal()* for Series or DataFrames with differing unicode data (GH20503)

**Contributors**

A total of 328 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Aaron Critchley
• AbdealiJK +
• Adam Hooper +
• Albert Villanova del Moral
• Alejandro Giacometti +
• Alejandro Hohmann +
• Alex Rychyk
• Alexander Buchkovsky
• Alexander Lenail +
• Alexander Michael Schade
• Aly Sivji +
• Andreas Kölтрinger +
• Andrew
• Andrew Bui +
• András Novoszáth +
• Andy Craze +
• Andy R. Terrel
• Anh Le +
• Anil Kumar Pallekonda +
• Antoine Pitrou +
• Antonio Linde +
• Antonio Molina +
• Antonio Quinonez +
• Armin Varshokar +
• Artem Bogachev +
• Avi Sen +
• Azeez Oluwafemi +
• Ben Auffarth +
• Bernhard Thiel +
• Bhavesh Poddar +
• BielStela +
• Blair +
• Bob Haffner
• Brett Naul +
• Brock Mendel
• Bryce Guinta +
• Carlos Eduardo Moreira dos Santos +
• Carlos García Márquez +
• Carol Willing
• Cheuk Ting Ho +
• Chitrank Dixit +
• Chris
• Chris Burr +
• Chris Catalfo +
• Chris Mazzullo
• Christian Chwala +
• Cihan Ceyhan +
• Clemens Brunner
• Colin +
• Cornelius Riemenschneider
• Crystal Gong +
• DaanVanHauwermeiren
• Dan Dixey +
• Daniel Frank +
• Daniel Garrido +
• Daniel Sakuma +
• DataOmbudsman +
• Dave Hirschfeld
• Dave Lewis +
• David Adrián Cañones Castellano +
• David Arcos +
• David C Hall +
• David Fischer
• David Hoese +
• David Lutz +
• David Polo +
• David Stansby
• Dennis Kamau +
• Dillon Niederhut
• Dimitri +
• Dr. Irv
• Dror Atariah
• Eric Chea +
• Eric Kisslinger
• Eric O. LEBIGOT (EOL) +
• FAN-GOD +
• Fabian Retkowski +
• Fer Sar +
• Gabriel de Maeztu +
• Gianpaolo Macario +
• Giftlin Rajaiah
• Gilberto Olimpio +
• Gina +
• Gjelt +
• Graham Inggs +
• Grant Roch
• Grant Smith +
• Grzegorz Konefał +
• Guilherme Beltramini
• HagaiHargil +
• Hamish Pitkeathly +
• Hammad Mashkoor +
• Hannah Ferchland +
• Hans
• Haochen Wu +
• Hissashi Rocha +
• Iain Barr +
• Ibrahim Sharaf ElDen +
• Ignasi Fosch +
• Igor Conrado Alves de Lima +
• Igor Shelvinskyi +
• Imanflow +
• Ingolf Becker
• Israel Saeta Pérez
• Iva Koevska +
• Jakub Nowacki +
• Jan F-F +
• Jan Koch +
• Jan Werkmann
• Janelle Zoutkamp +
• Jason Bandlow +
• Jaume Bonet +
• Jay Alammar +
• Jeff Reback
• JennaVergeynst
• Jimmy Woo +
• Jing Qiang Goh +
• Joachim Wagner +
• Joan Martin Miralles +
• Joel Nothman
• Joeun Park +
• John Cant +
• Johnny Metz +
• Jon Mease
• Jonas Schulze +
• Jongwony +
• Jordi Contestí +
• Joris Van den Bossche
• José F. R. Fonseca +
• Jovixe +
• Julio Martinez +
• Jörg Döpfert
• KOBAYASHI Itoku +
• Kate Surta +
• Kenneth +
• Kevin Kuhl
• Kevin Sheppard
• Krzysztof Chomski
• Ksenia +
• Ksenia Bobrova +
• Kunal Gosar +
• Kurtis Kerstein +
• Kyle Barron +
• Laksh Arora +
• Laurens Geffert +
• Leif Walsh
• Liam Marshall +
• Liam3851 +
• Licht Takeuchi
• Liudmila +
• Ludovico Russo +
• Mabel Villalba +
• Manan Pal Singh +
• Manraj Singh
• Marc +
• Marc Garcia
• Marco Hemken +
• Maria del Mar Bibiloni +
• Mario Corchero +
• Mark Woodbridge +
• Martin Journois +
• Mason Gallo +
• Matias Heikkilä +
• Matt Braymer-Hayes
• Matt Kirk +
• Matt Maybeno +
• Matthew Kirk +
• Matthew Rocklin +
• Matthew Roeschke
• Matthias Bussonnier +
• Max Mikhaylov +
• Maxim Veksler +
• Maximilian Roos
• Maximiliano Greco +
• Michael Penkov
• Michael Röttger +
• Michael Selik +
• Michael Waskom
• Mie~~~
• Mike Kutzma +
• Ming Li +
• Mitar +
• Mitch Negus +
• Montana Low +
• Moritz Münst +
• Mortada Mehyar
• Myles Braithwaite +
• Nate Yoder
• Nicholas Ursa +
• Nick Chmura
• Nikos Karagiannakis +
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Nipun Sadvilkar +
- Nis Martensen +
- Noah +
- Noémi Éltető +
- Olivier Bilodeau +
- Ondrej Kokes +
- Onno Eberhard +
- Paul Ganssle +
- Paul Mannino +
- Paul Reidy
- Paulo Roberto de Oliveira Castro +
- Pepe Flores +
- Peter Hoffmann
- Phil Ngo +
- Pietro Battiston
- Pranav Suri +
- Priyanka Ojha +
- Pulkit Maloo +
- README Bot +
- Ray Bell +
- Riccardo Magliocchetti +
- Ridhwan Luthra +
- Robert Meyer
- Robin
- Robin Kiplang’at +
- Rohan Pandit +
- Rok Mihevc +
- Rouz Azari
- Ryszard T. Kaleta +
- Sam Cohan
- Sam Foo
- Samir Musali +
- Samuel Sinayoko +
- Sangwoong Yoon
- SarahJessica +
- Sharad Vijalapuram +
• Shubham Chaudhary +
• SiYoungOh +
• Sietse Brouwer
• Simone Basso +
• Stefania Delprete +
• Stefano Cianciulli +
• Stephen Childs +
• StephenVoland +
• Stijn Van Hoey +
• Sven
• Talitha Pumar +
• Tarbo Fukazawa +
• Ted Petrou +
• Thomas A Caswell
• Tim Hoffmann +
• Tim Swast
• Tom Augspurger
• Tommy +
• Tulio Casagrande +
• Tushar Gupta +
• Tushar Mittal +
• Upkar Lidder +
• Victor Villas +
• Vince W +
• Vinícius Figueiredo +
• Vipin Kumar +
• WBare
• Wenhuan +
• Wes Turner
• William Ayd
• Wilson Lin +
• Xbar
• Yaroslav Halchenko
• Yee Mey
• Yeongseon Choe +
• Yian +
• Yimeng Zhang
• ZhuBaohe +
• Zihao Zhao +
• adatasetaday +
• akielbowicz +
• akosel +
• alinde1 +
• amuta +
• bolkedebriu
• cbertinato
• cgohlke
• charlie0389 +
• chris-b1
• csfarkas +
• dajcs +
• deflatSOCO +
• derestle-htwg
• discort
• dmanikowski-reef +
• donK23 +
• elrubio +
• fivemok +
• fjiod
• fjetter +
• froesser +
• gabrielclow
• gyoung
• ghasemnaddaf
• h-vetinari +
• himanshu awasthi +
• ignamv +
• jayfoad +
• jazzmuesli +
• jbrockmendel
• jen w +
• jjames34 +
• joaoavf +
• joders +
• jschendel
• juan huguet +
• l736x +
• luzpaz +
• mdeboc +
• miguelmorin +
• miker985
• miquelcamprodon +
• orereta +
•ottiP +
• peterpanmj +
• rafarui +
• raph-m +
• readyready15728 +
• rmihael +
• samghelms +
• scriptomation +
• sfoo +
• stefansimik +
• stonebig
• tmnhat2001 +
• tomneep +
• topper-123
• tv3141 +
• verakai +
• xpvpc +
• zhanghui +
5.9 Version 0.22

5.9.1 Version 0.22.0 (December 29, 2017)

This is a major release from 0.21.1 and includes a single, API-breaking change. We recommend that all users upgrade to this version after carefully reading the release note (singular!).

Backwards incompatible API changes

pandas 0.22.0 changes the handling of empty and all-NA sums and products. The summary is that

- The sum of an empty or all-NA Series is now 0
- The product of an empty or all-NA Series is now 1
- We’ve added a min_count parameter to .sum() and .prod() controlling the minimum number of valid values for the result to be valid. If fewer than min_count non-NA values are present, the result is NA. The default is 0. To return NaN, the 0.21 behavior, use min_count=1.

Some background: In pandas 0.21, we fixed a long-standing inconsistency in the return value of all-NA series depending on whether or not bottleneck was installed. See Sum/prod of all-NaN or empty Series/DataFrames is now consistently NaN. At the same time, we changed the sum and prod of an empty Series to also be NaN.

Based on feedback, we’ve partially reverted those changes.

Arithmetic operations

The default sum for empty or all-NA Series is now 0.

*pandas 0.21.x*

```
In [1]: pd.Series([]).sum()
Out[1]: nan

In [2]: pd.Series([np.nan]).sum()
Out[2]: nan
```

*pandas 0.22.0*

```
In [1]: pd.Series([]).sum()
Out[1]: 0.0

In [2]: pd.Series([np.nan]).sum()
Out[2]: 0.0
```

The default behavior is the same as pandas 0.20.3 with bottleneck installed. It also matches the behavior of NumPy’s np.nansum on empty and all-NA arrays.

To have the sum of an empty series return NaN (the default behavior of pandas 0.20.3 without bottleneck, or pandas 0.21.x), use the min_count keyword.

```
In [3]: pd.Series([]).sum(min_count=1)
Out[3]: nan
```

Thanks to the skipna parameter, the .sum on an all-NA series is conceptually the same as the .sum of an empty one with skipna=True (the default).
The `min_count` parameter refers to the minimum number of non-null values required for a non-NA sum or product. `Series.prod()` has been updated to behave the same as `Series.sum()`, returning 1 instead.

These changes affect `Dataframe.sum()` and `Dataframe.prod()` as well. Finally, a few less obvious places in pandas are affected by this change.

### Grouping by a Categorical

Grouping by a Categorical and summing now returns 0 instead of NaN for categories with no observations. The product now returns 1 instead of NaN.

**pandas 0.21.x**

```python
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])

In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
   a    3.0
   b   NaN
dtype: float64
```

**pandas 0.22**

```python
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])

In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
   a    3
   b    0
Length: 2, dtype: int64
```

To restore the 0.21 behavior of returning NaN for unobserved groups, use `min_count>=1`.

```python
In [10]: pd.Series([1, 2]).groupby(grouper).sum(min_count=1)
Out[10]:
   a    3.0
   b   NaN
Length: 2, dtype: float64
```
Resample

The sum and product of all-NA bins has changed from NaN to 0 for sum and 1 for product.

**pandas 0.21.x**

```python
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
                  index=pd.date_range('2017', periods=4))

Out[11]:
2017-01-01 1.0
2017-01-02 1.0
2017-01-03 NaN
2017-01-04 NaN
Freq: D, dtype: float64

In [12]: s.resample('2d').sum()

Out[12]:
2017-01-01 2.0
2017-01-03 NaN
Freq: 2D, dtype: float64
```

**pandas 0.22.0**

```python
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
                  index=pd.date_range("2017", periods=4))

In [12]: s.resample("2d").sum()

Out[12]:
2017-01-01 2.0
2017-01-03 0.0
Freq: 2D, Length: 2, dtype: float64
```

To restore the 0.21 behavior of returning NaN, use min_count>=1.

```python
In [13]: s.resample("2d").sum(min_count=1)

Out[13]:
2017-01-01 2.0
2017-01-03 NaN
Freq: 2D, Length: 2, dtype: float64
```

In particular, upsampling and taking the sum or product is affected, as upsampling introduces missing values even if the original series was entirely valid.

**pandas 0.21.x**

```python
In [14]: idx = pd.DatetimeIndex(["2017-01-01", "2017-01-02"])

In [15]: pd.Series([1, 2], index=idx).resample('12H').sum()

Out[15]:
2017-01-01 00:00:00 1.0
2017-01-01 12:00:00 NaN
2017-01-02 00:00:00 2.0
Freq: 12H, dtype: float64
```

**pandas 0.22.0**

""
In [14]: idx = pd.DatetimeIndex(['2017-01-01', '2017-01-02'])

In [15]: pd.Series([1, 2], index=idx).resample('12H').sum()
Out[15]:
2017-01-01 00:00:00 1
2017-01-01 12:00:00 0
2017-01-02 00:00:00 2
Freq: 12H, Length: 3, dtype: int64

Once again, the `min_count` keyword is available to restore the 0.21 behavior.

In [16]: pd.Series([1, 2], index=idx).resample('12H').sum(min_count=1)
Out[16]:
2017-01-01 00:00:00 1.0
2017-01-01 12:00:00 NaN
2017-01-02 00:00:00 2.0
Freq: 12H, Length: 3, dtype: float64

**Rolling and expanding**

Rolling and expanding already have a `min_periods` keyword that behaves similar to `min_count`. The only case that changes is when doing a rolling or expanding sum with `min_periods=0`. Previously this returned `NaN`, when fewer than `min_periods` non-NA values were in the window. Now it returns `0`.

*pandas 0.21.1*

In [17]: s = pd.Series([np.nan, np.nan])

In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0   NaN
1   NaN
dtype: float64

*pandas 0.22.0*

In [17]: s = pd.Series([np.nan, np.nan])

In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0   0.0
1   0.0
Length: 2, dtype: float64

The default behavior of `min_periods=None`, implying that `min_periods` equals the window size, is unchanged.

5.9. Version 0.22
Compatibility

If you maintain a library that should work across pandas versions, it may be easiest to exclude pandas 0.21 from your requirements. Otherwise, all your `sum()` calls would need to check if the `Series` is empty before summing.

With setuptools, in your `setup.py` use:

```python
install_requires=['pandas!=0.21.*', ...]
```

With conda, use

```yaml
requirements:
  run:
    - pandas !=0.21.*, !=0.21.1
```

Note that the inconsistency in the return value for all-NA series is still there for pandas 0.20.3 and earlier. Avoiding pandas 0.21 will only help with the empty case.

Contributors

A total of 1 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Tom Augspurger

5.10 Version 0.21

5.10.1 Version 0.21.1 (December 12, 2017)

This is a minor bug-fix release in the 0.21.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Temporarily restore matplotlib datetime plotting functionality. This should resolve issues for users who implicitly relied on pandas to plot datetimes with matplotlib. See [here](#).
- Improvements to the Parquet IO functions introduced in 0.21.0. See [here](#).

What's new in v0.21.1

- *Restore Matplotlib datetime converter registration*
- *New features*
  - *Improvements to the Parquet IO functionality*
  - *Other enhancements*
- *Deprecations*
- *Performance improvements*
- *Bug fixes*
  - *Conversion*
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Indexing
- IO
- Plotting
- GroupBy/resample/rolling
- Reshaping
- Numeric
- Categorical
- String
- Contributors

Restore Matplotlib datetime converter registration

pandas implements some matplotlib converters for nicely formatting the axis labels on plots with datetime or Period values. Prior to pandas 0.21.0, these were implicitly registered with matplotlib, as a side effect of import pandas.

In pandas 0.21.0, we required users to explicitly register the converter. This caused problems for some users who relied on those converters being present for regular matplotlib.pyplot plotting methods, so we’re temporarily reverting that change; pandas 0.21.1 again registers the converters on import, just like before 0.21.0.

We’ve added a new option to control the converters: `pd.options.plotting.matplotlib.register_converters`. By default, they are registered. Toggling this to `False` removes pandas’ formatters and restore any converters we overwrote when registering them (GH18301).

We’re working with the matplotlib developers to make this easier. We’re trying to balance user convenience (automatically registering the converters) with import performance and best practices (importing pandas shouldn’t have the side effect of overwriting any custom converters you’ve already set). In the future we hope to have most of the datetime formatting functionality in matplotlib, with just the pandas-specific converters in pandas. We’ll then gracefully deprecate the automatic registration of converters in favor of users explicitly registering them when they want them.

New features

Improvements to the Parquet IO functionality

- `DataFrame.to_parquet()` will now write non-default indexes when the underlying engine supports it. The indexes will be preserved when reading back in with `read_parquet()` (GH18581).
- `read_parquet()` now allows to specify the columns to read from a parquet file (GH18154)
- `read_parquet()` now allows to specify kwargs which are passed to the respective engine (GH18216)
Other enhancements

- `Timestamp.timestamp()` is now available in Python 2.7. (GH17329)
- `Grouper` and `TimeGrouper` now have a friendly repr output (GH18203).

Deprecations

- `pandas.tseries.register` has been renamed to `pandas.plotting.register_matplotlib_converters()` (GH18301)

Performance improvements

- Improved performance of plotting large series/dataframes (GH18236).

Bug fixes

Conversion

- Bug in `TimedeltaIndex` subtraction could incorrectly overflow when NaT is present (GH17791)
- Bug in `DatetimeIndex` subtracting datetimelike from DatetimeIndex could fail to overflow (GH18020)
- Bug in `IntervalIndex.copy()` when copying and `IntervalIndex` with non-default `closed` (GH18339)
- Bug in `DataFrame.to_dict()` where columns of datetime that are tz-aware were not converted to required arrays when used with `orient='records'`, raising `TypeError` (GH18372)
- Bug in `DateTimeIndex` and `date_range()` where mismatching tz-aware start and end timezones would not raise an err if `end.tzinfo` is None (GH18431)
- Bug in `Series.fillna()` which raised when passed a long integer on Python 2 (GH18159).

Indexing

- Bug in a boolean comparison of a `datetime.datetime` and a `datetime64[ns]` dtype Series (GH17965)
- Bug where a `MultiIndex` with more than a million records was not raising `AttributeError` when trying to access a missing attribute (GH18165)
- Bug in `IntervalIndex` constructor when a list of intervals is passed with non-default `closed` (GH18334)
- Bug in `Index.putmask` when an invalid mask passed (GH18368)
- Bug in masked assignment of a `timedelta64[ns]` dtype Series, incorrectly coerced to float (GH18493)
• Bug in class:~pandas.io.stata.StataReader not converting date/time columns with display formatting addressed (GH17990). Previously columns with display formatting were normally left as ordinal numbers and not converted to datetime objects.

• Bug in `read_csv()` when reading a compressed UTF-16 encoded file (GH18071)

• Bug in `read_csv()` for handling null values in index columns when specifying `na_filter=False` (GH5239)

• Bug in `read_csv()` when reading numeric category fields with high cardinality (GH18186)

• Bug in `DataFrame.to_csv()` when the table had MultiIndex columns, and a list of strings was passed in for `header` (GH5539)

• Bug in parsing integer datetime-like columns with specified format in `read_sql` (GH17855).

• Bug in `DataFrame.to_msgpack()` when serializing data of the `numpy.bool_` datatype (GH18390)

• Bug in `read_json()` not decoding when reading line delimited JSON from S3 (GH17200)

• Bug in `pandas.io.json.json_normalize()` to avoid modification of `meta` (GH18610)

• Bug in `to_latex()` where repeated MultiIndex values were not printed even though a higher level index differed from the previous row (GH14484)

• Bug when reading NaN-only categorical columns in HDFStore (GH18413)

• Bug in `DataFrame.to_latex()` with `longtable=True` where a latex multicolumn always spanned over three columns (GH17959)

Plotting

• Bug in `DataFrame.plot()` and `Series.plot()` with `DatetimeIndex` where a figure generated by them is not pickleable in Python 3 (GH18439)

GroupBy/resample/rolling

• Bug in `DataFrame.resample(...).apply(...)` when there is a callable that returns different columns (GH15169)

• Bug in `DataFrame.resample(...)` when there is a time change (DST) and resampling frequency is 12h or higher (GH15549)

• Bug in `pd.DataFrameGroupBy.count()` when counting over a datetimelike column (GH13393)

• Bug in `rolling.var` where calculation is inaccurate with a zero-valued array (GH18430)

Reshaping

• Error message in `pd.merge_asof()` for key datatype mismatch now includes datatype of left and right key (GH18068)

• Bug in `pd.concat` when empty and non-empty DataFrames or Series are concatenated (GH18178 GH18187)

• Bug in `DataFrame.filter(...)` when unicode is passed as a condition in Python 2 (GH13101)

• Bug when merging empty DataFrames when `np.seterr(divide='raise')` is set (GH17776)
Numeric

- Bug in `pd.Series.rolling.skew()` and `rolling.kurt()` with all equal values has floating issue (GH18044)

Categorical

- Bug in `DataFrame.astype()` where casting to 'category' on an empty `DataFrame` causes a segmentation fault (GH18004)
- Error messages in the testing module have been improved when items have different `CategoricalDtype` (GH18069)
- `CategoricalIndex` can now correctly take a `pd.api.types.CategoricalDtype` as its dtype (GH18116)
- Bug in `Categorical.unique()` returning read-only codes array when all categories were `NaN` (GH18051)
- Bug in `DataFrame.groupby(axis=1)` with a `CategoricalIndex` (GH18432)

String

- `Series.str.split()` will now propagate `NaN` values across all expanded columns instead of `None` (GH18450)

Contributors

A total of 46 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Critchley +
- Alex Rychyk
- Alexander Buchkovsky +
- Alexander Michael Schade +
- Chris Mazzullo
- Cornelius Riemenschneider +
- Dave Hirschfeld +
- David Fischer +
- David Stansby +
- Dror Atariah +
- Eric Kisslinger +
- Hans +
- Ingolf Becker +
- Jan Werkmann +
- Jeff Reback
- Joris Van den Bossche
- Jörg Döpfert +
- Kevin Kuhl +
• Krzysztof Chomski +
• Leif Walsh
• Licht Takeuchi
• Manraj Singh +
• Matt Braymer-Hayes +
• Michael Waskom +
• Mie~~~ +
• Peter Hoffmann +
• Robert Meyer +
• Sam Cohan +
• Sietse Brouwer +
• Sven +
• Tim Swast
• Tom Augspurger
• Wes Turner
• William Ayd +
• Yee Mey +
• bolkedebruin +
• cgholk
• derestle-htwg +
• fjiod +
• gabrielclow +
• gfyoung
• ghasemnaddaf +
• jbrockmendel
• jschendel
• miker985 +
• topper-123

5.10.2 Version 0.21.0 (October 27, 2017)

This is a major release from 0.20.3 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

• Integration with Apache Parquet, including a new top-level `read_parquet()` function and `DataFrame.to_parquet()` method, see here.
• New user-facing pandas.api.types.CategoricalDtype for specifying categoricals independent of the data, see here.
• The behavior of sum and prod on all-NaN Series/DataFrames is now consistent and no longer depends on whether bottleneck is installed, and sum and prod on empty Series now return NaN instead of 0, see here.
• Compatibility fixes for pypy, see here.
• Additions to the drop, reindex and rename API to make them more consistent, see here.
• Addition of the new methods DataFrame.infer_objects (see here) and GroupBy.pipe (see here).
• Indexing with a list of labels, where one or more of the labels is missing, is deprecated and will raise a KeyError in a future version, see here.

Check the API Changes and deprecations before updating.

What’s new in v0.21.0

• New features
  – Integration with Apache Parquet file format
  – Method infer_objects type conversion
  – Improved warnings when attempting to create columns
  – Method drop now also accepts index/columns keywords
  – Methods rename, reindex now also accept axis keyword
  – CategoricalDtype for specifying categoricals
  – GroupBy objects now have a pipe method
  – Categorical.rename_categories accepts a dict-like
  – Other enhancements

• Backwards incompatible API changes
  – Dependencies have increased minimum versions
  – Sum/prod of all-NaN or empty Series/DataFrames is now consistently NaN
  – Indexing with a list with missing labels is deprecated
  – NA naming changes
  – Iteration of Series/Index will now return Python scalars
  – Indexing with a Boolean Index
  – PeriodIndex resampling
  – Improved error handling during item assignment in pd.eval
  – Dtype conversions
  – MultiIndex constructor with a single level
  – UTC localization with Series
  – Consistency of range functions
  – No automatic Matplotlib converters
  – Other API changes
• **Deprecations**
  – Series.select and DataFrame.select
  – Series.argmax and Series.argmin
• **Removal of prior version deprecations/changes**
• **Performance improvements**
• **Documentation changes**
• **Bug fixes**
  – Conversion
  – Indexing
  – IO
  – Plotting
  – GroupBy/resample/rolling
  – Sparse
  – Reshaping
  – Numeric
  – Categorical
  – PyPy
  – Other
• **Contributors**

---

**New features**

**Integration with Apache Parquet file format**

Integration with Apache Parquet, including a new top-level `read_parquet()` and `DataFrame.to_parquet()` method, see [here](GH15838, GH17438).

Apache Parquet provides a cross-language, binary file format for reading and writing data frames efficiently. Parquet is designed to faithfully serialize and de-serialize DataFrame s, supporting all of the pandas dtypes, including extension dtypes such as datetime with timezones.

This functionality depends on either the pyarrow or fastparquet library. For more details, see the IO docs on Parquet.

**Method `infer_objects` type conversion**

The `DataFrame.infer_objects()` and `Series.infer_objects()` methods have been added to perform dtype inference on object columns, replacing some of the functionality of the deprecated `convert_objects` method. See the documentation [here](GH11221) for more details.

This method only performs soft conversions on object columns, converting Python objects to native types, but not any coercive conversions. For example:
In [1]: df = pd.DataFrame({
'A': [1, 2, 3],
'B': np.array([1, 2, 3], dtype='object'),
'C': ['1', '2', '3']})

In [2]: df.dtypes
Out[2]:
A int64
B object
C object
Length: 3, dtype: object

In [3]: df.infer_objects().dtypes
Out[3]:
A int64
B int64
C object
Length: 3, dtype: object

Note that column 'C' was not converted - only scalar numeric types will be converted to a new type. Other types of
conversion should be accomplished using the to_numeric() function (or to_datetime(), to_timedelta()).

In [4]: df = df.infer_objects()
In [5]: df['C'] = pd.to_numeric(df['C'], errors='coerce')

In [6]: df.dtypes
Out[6]:
A int64
B int64
C int64
Length: 3, dtype: object

Improved warnings when attempting to create columns

New users are often puzzled by the relationship between column operations and attribute access on DataFrame in-
stances (GH7175). One specific instance of this confusion is attempting to create a new column by setting an attribute
on the DataFrame:

In [1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In [2]: df.two = [4, 5, 6]

This does not raise any obvious exceptions, but also does not create a new column:

In [3]: df
Out[3]:
   one
0  1.0
1  2.0
2  3.0

Setting a list-like data structure into a new attribute now raises a UserWarning about the potential for unexpected
behavior. See Attribute Access.
Method drop now also accepts index/columns keywords

The drop() method has gained index/columns keywords as an alternative to specifying the axis. This is similar to the behavior of reindex (GH12392).

For example:

```python
In [7]: df = pd.DataFrame(np.arange(8).reshape(2, 4),
   ...:         columns=['A', 'B', 'C', 'D'])

In [8]: df
Out[8]:
   A  B  C  D
0  0  1  2  3
1  4  5  6  7

[2 rows x 4 columns]

In [9]: df.drop(['B', 'C'], axis=1)
Out[9]:
   A  D
0  0  3
1  4  7

[2 rows x 2 columns]

# the following is now equivalent
In [10]: df.drop(columns=['B', 'C'])
Out[10]:
   A  D
0  0  3
1  4  7

[2 rows x 2 columns]
```

Methods rename, reindex now also accept axis keyword

The DataFrame.rename() and DataFrame.reindex() methods have gained the axis keyword to specify the axis to target with the operation (GH12392).

Here's rename:

```python
In [11]: df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

In [12]: df.rename(str.lower, axis='columns')
Out[12]:
   a  b
0  1  4
1  2  5
2  3  6

[3 rows x 2 columns]
```
In [13]: df.rename(id, axis='index')
Out[13]:

    A  B
0 94911568096480 1  4
1 94911568096512 2  5
2 94911568096544 3  6

[3 rows x 2 columns]

And reindex:

In [14]: df.reindex(['A', 'B', 'C'], axis='columns')
Out[14]:

    A  B  C
0   1  4  NaN
1   2  5  NaN
2   3  6  NaN

[3 rows x 3 columns]

In [15]: df.reindex([0, 1, 3], axis='index')
Out[15]:

     A  B
0  1.0  4.0
1  2.0  5.0
3  NaN  NaN

[3 rows x 2 columns]

The “index, columns” style continues to work as before.

In [16]: df.rename(index=id, columns=str.lower)
Out[16]:

    a  b
0 94911568096480 1  4
1 94911568096512 2  5
2 94911568096544 3  6

[3 rows x 2 columns]

In [17]: df.reindex(index=[0, 1, 3], columns=['A', 'B', 'C'])
Out[17]:

    A  B  C
0  1.0  4.0  NaN
1  2.0  5.0  NaN
3  NaN  NaN  NaN

[3 rows x 3 columns]

We highly encourage using named arguments to avoid confusion when using either style.
CategoricalDtype for specifying categoricals

pandas.api.types.CategoricalDtype has been added to the public API and expanded to include the categories and ordered attributes. A CategoricalDtype can be used to specify the set of categories and orderedness of an array, independent of the data. This can be useful for example, when converting string data to a Categorical (GH14711, GH15078, GH16015, GH17643):

```python
In [18]: from pandas.api.types import CategoricalDtype

In [19]: s = pd.Series(['a', 'b', 'c', 'a'])  # strings

In [20]: dtype = CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)

In [21]: s.astype(dtype)
Out[21]:
0   a
1   b
2   c
3   a
Length: 4, dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']
```

One place that deserves special mention is in `read_csv()`. Previously, with `dtype={'col': 'category'}`, the returned values and categories would always be strings.

```python
In [22]: data = 'A,B
1
2
3'

In [23]: pd.read_csv(StringIO(data), dtype={'B': 'category'}).B.cat.categories
Out[23]: Index([1, 2, 3], dtype='object')
```

Notice the “object” dtype.

With a CategoricalDtype of all numerics, datetimes, or timedeltas, we can automatically convert to the correct type

```python
In [24]: dtype = {'B': CategoricalDtype([1, 2, 3])}

In [25]: pd.read_csv(StringIO(data), dtype=dtype).B.cat.categories
Out[25]: Int64Index([1, 2, 3], dtype='int64')
```

The values have been correctly interpreted as integers.

The .dtype property of a Categorical, CategoricalIndex or a Series with categorical type will now return an instance of CategoricalDtype. While the repr has changed, str(CategoricalDtype()) is still the string 'category'. We’ll take this moment to remind users that the preferred way to detect categorical data is to use pandas.api.types.is_categorical_dtype(), and not str(dtype) == 'category'.

See the CategoricalDtype docs for more.
GroupBy objects now have a pipe method

GroupBy objects now have a pipe method, similar to the one on DataFrame and Series, that allow for functions that take a GroupBy to be composed in a clean, readable syntax. (GH17871)

For a concrete example on combining .groupby and .pipe, imagine having a DataFrame with columns for stores, products, revenue and sold quantity. We’d like to do a groupwise calculation of prices (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable.

First we set the data:

```
In [26]: import numpy as np
In [27]: n = 1000
In [28]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
    ....:     'Product': np.random.choice(['Product_1',
    ....:         'Product_2',
    ....:     'Product_3'], n),
    ....:     'Revenue': (np.random.rand(n) * 50 + 10).round(2),
    ....:     'Quantity': np.random.randint(1, 10, size=n)})
```

```
In [29]: df.head(2)
Out[29]:
     Store   Product  Revenue  Quantity
0  Store_2  Product_2    32.09      7
1  Store_1  Product_3    14.20      1
```

Now, to find prices per store/product, we can simply do:

```
In [30]: (df.groupby(['Store', 'Product'])
    ....:     .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
    ....:     .unstack().round(2))
```

```
In [30]:
Out[30]:
     Product
Store     Product_1  Product_2  Product_3
Store_1    6.730      6.720      7.140
Store_2    7.589      6.980      7.230
```

See the documentation for more.
Categorical.rename_categories accepts a dict-like

rename_categories() now accepts a dict-like argument for new_categories. The previous categories are looked up in the dictionary's keys and replaced if found. The behavior of missing and extra keys is the same as in DataFrame.rename().

```python
In [31]: c = pd.Categorical(['a', 'a', 'b'])

In [32]: c.rename_categories({'a': 'eh', 'b': 'bee'})
Out[32]:
['eh', 'eh', 'bee']
Categories (2, object): ['eh', 'bee']
```

**Warning:** To assist with upgrading pandas, rename_categories treats Series as list-like. Typically, Series are considered to be dict-like (e.g. in .rename, .map). In a future version of pandas rename_categories will change to treat them as dict-like. Follow the warning message's recommendations for writing future-proof code.

```python
In [33]: c.rename_categories(pd.Series([0, 1], index=['a', 'c']))
FutureWarning: Treating Series 'new_categories' as a list-like and using the values.
In a future version, 'rename_categories' will treat Series like a dictionary.
For dict-like, use 'new_categories.to_dict()'
For list-like, use 'new_categories.values'.
Out[33]:
[0, 0, 1]
Categories (2, int64): [0, 1]
```

Other enhancements

New functions or methods

- nearest() is added to support nearest-neighbor upsampling (GH17496).
- Index has added support for a to_frame method (GH15230).

New keywords

- Added a skipna parameter to infer_dtype() to support type inference in the presence of missing values (GH17059).
- Series.to_dict() and DataFrame.to_dict() now support an into keyword which allows you to specify the collections.Mapping subclass that you would like returned. The default is dict, which is backwards compatible. (GH16122)
- Series.set_axis() and DataFrame.set_axis() now support the inplace parameter. (GH14636)
- Series.to_pickle() and DataFrame.to_pickle() have gained a protocol parameter (GH16252). By default, this parameter is set to HIGHEST_PROTOCOL
- read_feather() has gained the nthreads parameter for multi-threaded operations (GH16359)
- DataFrame.clip() and Series.clip() have gained an inplace argument. (GH15388)
• *crosstab()* has gained a `margins_name` parameter to define the name of the row / column that will contain the totals when `margins=True`. (GH15972)

• *read_json()* now accepts a `chunksize` parameter that can be used when `lines=True`. If `chunksize` is passed, read_json now returns an iterator which reads in `chunksize` lines with each iteration. (GH17048)

• *read_json()* and *to_json()* now accept a compression argument which allows them to transparently handle compressed files. (GH17798)

**Various enhancements**

• Improved the import time of pandas by about 2.25x. (GH16764)

• Support for PEP 519 – Adding a file system path protocol on most readers (e.g. *read_csv()* and writers (e.g. *DataFrame.to_csv()*). (GH13823)

• Added a `__fspath__` method to *pd.HDFStore*, *pd.ExcelFile*, and *pd.ExcelWriter* to work properly with the file system path protocol (GH13823).

• The `validate` argument for *merge()* now checks whether a merge is one-to-one, one-to-many, many-to-one, or many-to-many. If a merge is found to not be an example of specified merge type, an exception of type *MergeError* will be raised. For more, see here (GH16270)

• Added support for PEP 518 (`pyproject.toml`) to the build system (GH16745)

• *RangeIndex.append()* now returns a *RangeIndex* object when possible (GH16212)

• *Series.rename_axis()* and *DataFrame.rename_axis()* with `inplace=True` now return `None` while renaming the axis inplace. (GH15704)

• *api.types.infer_dtype()* now infers decimals. (GH15690)

• *DataFrame.select_dtypes()* now accepts scalar values for include/exclude as well as list-like. (GH16855)

• *date_range()* now accepts ‘YS’ in addition to ‘AS’ as an alias for start of year. (GH9313)

• *date_range()* now accepts ‘Y’ in addition to ‘A’ as an alias for end of year. (GH9313)

• *DataFrame.add_prefix()* and *DataFrame.add_suffix()* now accept strings containing the ‘%’ character. (GH17151)

• Read/write methods that infer compression (*read_csv()* / *read_table()* / *read_pickle()* / *to_pickle()* can now infer from path-like objects, such as `pathlib.Path`. (GH17206)

• *read_sas()* now recognizes much more of the most frequently used date (datetime) formats in SAS7BDAT files. (GH15871)

• *DataFrame.items()* and *Series.items()* are now present in both Python 2 and 3 and is lazy in all cases. (GH13918, GH17213)

• *pandas.io.formats.style.Styler.where()* has been implemented as a convenience for *pandas.io.formats.style.Styler.applymap*. (GH17474)

• *MultiIndex.is_monotonic_decreasing()* has been implemented. Previously returned *False* in all cases. (GH16554)

• *read_excel()* raises *ImportError* with a better message if `xlrd` is not installed. (GH17613)

• *DataFrame.assign()* will preserve the original order of `**kwargs` for Python 3.6+ users instead of sorting the column names. (GH14207)

• *Series.reindex()* / *DataFrame.reindex()* / *Index.get_indexer()* now support list-like argument for tolerance. (GH17367)
**Backwards incompatible API changes**

**Dependencies have increased minimum versions**

We have updated our minimum supported versions of dependencies (GH15206, GH15543, GH15214). If installed, we now require:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimum Version</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy</td>
<td>1.9.0</td>
<td>X</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>1.4.3</td>
<td></td>
</tr>
<tr>
<td>Scipy</td>
<td>0.14.0</td>
<td></td>
</tr>
<tr>
<td>Bottleneck</td>
<td>1.0.0</td>
<td></td>
</tr>
</tbody>
</table>

Additionally, support has been dropped for Python 3.4 (GH15251).

**Sum/prod of all-NaN or empty Series/DataFrames is now consistently NaN**

*Note:* The changes described here have been partially reverted. See the v0.22.0 Whatsnew for more.

The behavior of `sum` and `prod` on all-NaN Series/DataFrames no longer depends on whether `bottleneck` is installed, and return value of `sum` and `prod` on an empty Series has changed (GH9422, GH15507).

Calling `sum` or `prod` on an empty or all-NaN Series, or columns of a DataFrame, will result in NaN. See the docs.

```
In [33]: s = pd.Series([np.nan])

Previously WITHOUT bottleneck installed:

In [2]: s.sum()
Out[2]: np.nan

Previously WITH bottleneck:

In [2]: s.sum()
Out[2]: 0.0

New behavior, without regard to the bottleneck installation:

In [34]: s.sum()
Out[34]: 0.0

Note that this also changes the sum of an empty Series. Previously this always returned 0 regardless of a bottleneck installation:

In [1]: pd.Series([]).sum()
Out[1]: 0

but for consistency with the all-NaN case, this was changed to return NaN as well:

In [35]: pd.Series([]).sum()
Out[35]: 0.0
```
Indexing with a list with missing labels is deprecated

Previously, selecting with a list of labels, where one or more labels were missing would always succeed, returning NaN for missing labels. This will now show a FutureWarning. In the future this will raise a KeyError (GH15747). This warning will trigger on a DataFrame or a Series for using .loc[] or [] when passing a list-of-labels with at least 1 missing label. See the deprecation docs.

```python
In [36]: s = pd.Series([1, 2, 3])

In [37]: s
Out[37]:
     0 1
     1 2
     2 3
Length: 3, dtype: int64
```

Previous behavior

```python
In [4]: s.loc[[1, 2, 3]]
Out[4]:
     1 2.0
     2 3.0
     3 NaN
dtype: float64
```

Current behavior

```python
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.
```

See the documentation here:

```python
Out[4]:
     1 2.0
     2 3.0
     3 NaN
dtype: float64
```

The idiomatic way to achieve selecting potentially not-found elements is via .reindex()

```python
In [38]: s.reindex([1, 2, 3])
Out[38]:
     1 2.0
     2 3.0
     3 NaN
Length: 3, dtype: float64
```

Selection with all keys found is unchanged.

```python
In [39]: s.loc[[1, 2]]
Out[39]:
     1 2
```

(continues on next page)
NA naming changes

In order to promote more consistency among the pandas API, we have added additional top-level functions `isna()` and `notna()` that are aliases for `isnull()` and `notnull()`. The naming scheme is now more consistent with methods like `.dropna()` and `.fillna()`. Furthermore in all cases where `.isnull()` and `.notnull()` methods are defined, these have additional methods named `.isna()` and `.notna()`, these are included for classes `Categorical`, `Index`, `Series`, and `DataFrame`. (GH15001).

The configuration option `pd.options.mode.use_inf_as_null` is deprecated, and `pd.options.mode.use_inf_as_na` is added as a replacement.

Iteration of Series/Index will now return Python scalars

Previously, when using certain iteration methods for a `Series` with dtype int or float, you would receive a numpy scalar, e.g. a `np.int64`, rather than a Python int. Issue (GH10904) corrected this for `Series.tolist()` and `list(Series)`. This change makes all iteration methods consistent, in particular, for `__iter__()` and `.map();` note that this only affects int/float dtypes. (GH13236, GH13258, GH14216).

```
In [40]: s = pd.Series([1, 2, 3])
In [41]: s
Out[41]:
0   1
1   2
2   3
Length: 3, dtype: int64

Previously:
In [2]: type(list(s)[0])
Out[2]: numpy.int64

New behavior:
In [42]: type(list(s)[0])
Out[42]: int
```

Furthermore this will now correctly box the results of iteration for `DataFrame.to_dict()` as well.

```
In [43]: d = {'a': [1], 'b': ['b']}
In [44]: df = pd.DataFrame(d)

Previously:
In [8]: type(df.to_dict()['a'][0])
Out[8]: numpy.int64

New behavior:
```
Indexing with a Boolean Index

Previously when passing a boolean Index to .loc, if the index of the Series/DataFrame had boolean labels, you would get a label based selection, potentially duplicating result labels, rather than a boolean indexing selection (where True selects elements), this was inconsistent how a boolean numpy array indexed. The new behavior is to act like a boolean numpy array indexer. (GH17738)

Previous behavior:

```python
In [46]: s = pd.Series([1, 2, 3], index=[False, True, False])
In [47]: s
Out[47]:
False 1
True  2
False 3
Length: 3, dtype: int64
```

```
In [59]: s.loc[pd.Index([True, False, True])]
Out[59]:
True  2
False 1
False 3
True  2
dtype: int64
```

Current behavior:

```python
In [48]: s.loc[pd.Index([True, False, True])]
Out[48]:
False 1
False 3
Length: 2, dtype: int64
```

Furthermore, previously if you had an index that was non-numeric (e.g. strings), then a boolean Index would raise a `KeyError`. This will now be treated as a boolean indexer.

Previously behavior:

```python
In [49]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
In [50]: s
Out[50]:
a  1
b  2
c  3
Length: 3, dtype: int64
```

```
In [39]: s.loc[pd.Index([True, False, True])]
KeyError: "None of [Index([True, False, True], dtype='object')] are in the [index]"
```
Current behavior

```python
In [51]: s.loc[pd.Index([True, False, True])]
Out[51]:
da   1
   c   3
Length: 2, dtype: int64
```

**PeriodIndex resampling**

In previous versions of pandas, resampling a Series/DataFrame indexed by a PeriodIndex returned a DatetimeIndex in some cases (GH12884). Resampling to a multiplied frequency now returns a PeriodIndex (GH15944). As a minor enhancement, resampling a PeriodIndex can now handle NaT values (GH13224)

Previous behavior:

```python
In [1]: pi = pd.period_range('2017-01', periods=12, freq='M')
In [2]: s = pd.Series(np.arange(12), index=pi)
In [3]: resampled = s.resample('2Q').mean()
In [4]: resampled
Out[4]:
2017-03-31    1.0
2017-09-30    5.5
2018-03-31   10.0
Freq: 2Q-DEC, dtype: float64

In [5]: resampled.index
Out[5]: DatetimeIndex(['2017-03-31', '2017-09-30', '2018-03-31'], dtype='datetime64[ns]', freq='2Q-DEC')
```

New behavior:

```python
In [52]: pi = pd.period_range('2017-01', periods=12, freq='M')
In [53]: s = pd.Series(np.arange(12), index=pi)
In [54]: resampled = s.resample('2Q').mean()
In [55]: resampled
Out[55]:
2017Q1   2.5
2017Q3   8.5
Freq: 2Q-DEC, Length: 2, dtype: float64

In [56]: resampled.index
Out[56]: PeriodIndex(['2017Q1', '2017Q3'], dtype='period[2Q-DEC]')
```

Upsampling and calling .ohlc() previously returned a Series, basically identical to calling .asfreq(). OHLC upsampling now returns a DataFrame with columns open, high, low and close (GH13083). This is consistent with downsampling and DatetimeIndex behavior.
Previous behavior:

```python
In [1]: pi = pd.period_range(start='2000-01-01', freq='D', periods=10)
In [2]: s = pd.Series(np.arange(10), index=pi)
In [3]: s.resample('H').ohlc()
Out[3]:
2000-01-01 00:00 0.0
... ...
2000-01-10 23:00 NaN
Freq: H, Length: 240, dtype: float64
In [4]: s.resample('M').ohlc()
Out[4]:
   open  high  low  close
2000-01 0 9 0 9
```

New behavior:

```python
In [57]: pi = pd.period_range(start='2000-01-01', freq='D', periods=10)
In [58]: s = pd.Series(np.arange(10), index=pi)
In [59]: s.resample('H').ohlc()
Out[59]:
   open  high  low  close
2000-01-01 00:00 0.0 0.0 0.0 0.0
2000-01-01 01:00 NaN NaN NaN NaN
... ...
2000-01-10 19:00 NaN NaN NaN NaN
2000-01-10 20:00 NaN NaN NaN NaN
2000-01-10 21:00 NaN NaN NaN NaN
2000-01-10 22:00 NaN NaN NaN NaN
2000-01-10 23:00 NaN NaN NaN NaN

[240 rows x 4 columns]
In [60]: s.resample('M').ohlc()
Out[60]:
   open  high  low  close
2000-01 0 9 0 9
[1 rows x 4 columns]
```
Improved error handling during item assignment in pd.eval

`eval()` will now raise a `ValueError` when item assignment malfunctions, or inplace operations are specified, but there is no item assignment in the expression (GH16732)

```python
In [61]: arr = np.array([1, 2, 3])
```

Previously, if you attempted the following expression, you would get a not very helpful error message:

```python
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...:
IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`)
and integer or boolean arrays are valid indices
```

This is a very long way of saying numpy arrays don’t support string-item indexing. With this change, the error message is now this:

```python
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...:
ValueError: Cannot assign expression output to target
```

It also used to be possible to evaluate expressions inplace, even if there was no item assignment:

```python
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
Out[4]: 3
```

However, this input does not make much sense because the output is not being assigned to the target. Now, a `ValueError` will be raised when such an input is passed in:

```python
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
...:
ValueError: Cannot operate inplace if there is no assignment
```

Dtype conversions

Previously assignments, `.where()` and `.fillna()` with a bool assignment, would coerce to the same type (e.g. int / float), or raise for datetimelikes. These will now preserve the bools with object dtypes. (GH16821).

```python
In [62]: s = pd.Series([1, 2, 3])
In [5]: s[1] = True
```

```python
In [6]: s
Out[6]:
0  1
1  1
2  3
dtype: int64
```

New behavior
Previously, as assignment to a datetimelike with a non-datetimelike would coerce the non-datetime-like item being assigned (GH14145).

In [65]: s = pd.Series([pd.Timestamp('2011-01-01'), pd.Timestamp('2012-01-01')])

In [1]: s[1] = 1
In [2]: s
Out[2]:
0 2011-01-01 00:00:00
1 1970-01-01 00:00:00
dtype: datetime64[ns]

These now coerce to object dtype.

In [66]: s[1] = 1
In [67]: s
Out[67]:
0 2011-01-01
1 1
Length: 2, dtype: object

• Inconsistent behavior in \texttt{.where()} with datetimelikes which would raise rather than coerce to object (GH16402)
• Bug in assignment against int64 data with \texttt{np.ndarray} with float64 dtype may keep int64 dtype (GH14001)

**MultiIndex constructor with a single level**

The \texttt{MultiIndex} constructors no longer squeezes a \texttt{MultiIndex} with all length-one levels down to a regular \texttt{Index}. This affects all the \texttt{MultiIndex} constructors. (GH17178)

Previous behavior:

In [2]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[2]: Index(['a', 'b'], dtype='object')

Length 1 levels are no longer special-cased. They behave exactly as if you had length 2+ levels, so a \texttt{MultiIndex} is always returned from all of the \texttt{MultiIndex} constructors:
UTC localization with Series

Previously, `to_datetime()` did not localize datetime series data when `utc=True` was passed. Now, `to_datetime()` will correctly localize series with a datetime64[ns, UTC] dtype to be consistent with how list-like and Index data are handled. (GH6415).

Previous behavior

```python
In [69]: s = pd.Series(['20130101 00:00:00'] * 3)
In [12]: pd.to_datetime(s, utc=True)
Out[12]:
0 2013-01-01
1 2013-01-01
2 2013-01-01
Length: 3, dtype: datetime64[ns]
```

New behavior

```python
In [70]: pd.to_datetime(s, utc=True)
Out[70]:
0 2013-01-01 00:00:00+00:00
1 2013-01-01 00:00:00+00:00
2 2013-01-01 00:00:00+00:00
Length: 3, dtype: datetime64[ns, UTC]
```

Additionally, DataFrames with datetime columns that were parsed by `read_sql_table()` and `read_sql_query()` will also be localized to UTC only if the original SQL columns were timezone aware datetime columns.

Consistency of range functions

In previous versions, there were some inconsistencies between the various range functions: `date_range()`, `bdate_range()`, `period_range()`, `timedelta_range()`, and `interval_range()`. (GH17471).

One of the inconsistent behaviors occurred when the `start`, `end` and `period` parameters were all specified, potentially leading to ambiguous ranges. When all three parameters were passed, `interval_range` ignored the `period` parameter, `period_range` ignored the `end` parameter, and the other range functions raised. To promote consistency among the range functions, and avoid potentially ambiguous ranges, `interval_range` and `period_range` will now raise when all three parameters are passed.

Previous behavior:

```python
In [2]: pd.interval_range(start=0, end=4, periods=6)
Out[2]:
IntervalIndex([(0, 1], (1, 2], (2, 3])
    closed='right',
    dtype='interval[int64]')
```

(continues on next page)
New behavior:

```python
In [2]: pd.interval_range(start=0, end=4, periods=6)
```

```
ValueError: Of the three parameters: start, end, and periods, exactly two must be specified
```

```python
In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
```

```
ValueError: Of the three parameters: start, end, and periods, exactly two must be specified
```

Additionally, the endpoint parameter `end` was not included in the intervals produced by `interval_range`. However, all other range functions include `end` in their output. To promote consistency among the range functions, `interval_range` will now include `end` as the right endpoint of the final interval, except if `freq` is specified in a way which skips `end`.

Previous behavior:

```python
In [4]: pd.interval_range(start=0, end=4)
```

```
Out[4]:
IntervalIndex([(0, 1], (1, 2], (2, 3]
            closed='right',
            dtype='interval[int64]')
```

New behavior:

```python
In [71]: pd.interval_range(start=0, end=4)
```

```
Out[71]: IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4]], dtype='interval[int64, right]')
```

### No automatic Matplotlib converters

pandas no longer registers our `date`, `time`, `datetime`, `datetime64`, and `Period` converters with matplotlib when pandas is imported. Matplotlib plot methods (`plt.plot`, `ax.plot`, ...), will not nicely format the x-axis for `DatetimeIndex` or `PeriodIndex` values. You must explicitly register these methods:

pandas built-in `Series.plot` and `DataFrame.plot` will register these converters on first-use (GH17710).

**Note:** This change has been temporarily reverted in pandas 0.21.1, for more details see [here](#).
Other API changes

- The Categorical constructor no longer accepts a scalar for the categories keyword. (GH16022)
- Accessing a non-existent attribute on a closed HDFStore will now raise an AttributeError rather than a ClosedFileError (GH16301)
- read_csv() now issues a UserWarning if the names parameter contains duplicates (GH17095)
- read_csv() now treats 'null' and 'n/a' strings as missing values by default (GH16471, GH16078)
- pandas.HDFStore's string representation is now faster and less detailed. For the previous behavior, use pandas.HDFStore.info(). (GH16503).
- Compression defaults in HDF stores now follow pytables standards. Default is no compression and if complib is missing and complevel > 0 zlib is used (GH15943)
- Index.get_indexer_non_unique() now returns a ndarray indexer rather than an Index; this is consistent with Index.get_indexer() (GH16819)
- Removed the @slow decorator from pandas._testing, which caused issues for some downstream packages’ test suites. Use @pytest.mark.slow instead, which achieves the same thing (GH16850)
- Moved definition of MergeError to the pandas.errors module.
- The signature of Series.set_axis() and DataFrame.set_axis() has been changed from set_axis(axis, labels) to set_axis(labels, axis=0), for consistency with the rest of the API. The old signature is deprecated and will show a FutureWarning (GH14636)
- Series.argmin() and Series.argmax() will now raise a TypeError when used with object dtypes, instead of a ValueError (GH13595)
- Period is now immutable, and will now raise an AttributeError when a user tries to assign a new value to the ordinal or freq attributes (GH17116).
- to_datetime() when passed a tz-aware origin= kwarg will now raise a more informative ValueError rather than a TypeError (GH16842)
- to_datetime() now raises a ValueError when format includes %W or %U without also including day of the week and calendar year (GH16774)
- Renamed non-functional index to index_col in read_stata() to improve API consistency (GH16342)
- Bug in DataFrame.drop() caused boolean labels False and True to be treated as labels 0 and 1 respectively when dropping indices from a numeric index. This will now raise a ValueError (GH16877)
- Restricted DateOffset keyword arguments. Previously, DateOffset subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176).

Deprecations

- DataFrame.from_csv() and Series.from_csv() have been deprecated in favor of read_csv() (GH4191)
- read_excel() has deprecated sheetname in favor of sheet_name for consistency with .to_excel() (GH10559).
- read_excel() has deprecated parse_cols in favor of usecols for consistency with read_csv() (GH4988)
- read_csv() has deprecated the tupleize_cols argument. Column tuples will always be converted to a MultiIndex (GH17060)
- DataFrame.to_csv() has deprecated the tupleize_cols argument. MultiIndex columns will be always written as rows in the CSV file (GH17060)
• The `convert` parameter has been deprecated in the `.take()` method, as it was not being respected (GH16948).

• `pd.options.html.border` has been deprecated in favor of `pd.options.display.html.border` (GH15793).

• `SeriesGroupBy.nth()` has deprecated `True` in favor of 'all' for its kwarg `dropna` (GH11038).

• `DataFrame.as_blocks()` is deprecated, as this is exposing the internal implementation (GH17302).

• `pd.TimeGrouper` is deprecated in favor of `pandas.Grouper` (GH16747).

• `cdate_range` has been deprecated in favor of `bdate_range()`, which has gained `weekmask` and `holidays` parameters for building custom frequency date ranges. See the documentation for more details (GH17596).

• passing `categories` or ordered `kwargs` to `Series.astype()` is deprecated, in favor of passing a `CategoricalDtype` (GH17636).

• `.get_value` and `.set_value` on `Series`, `DataFrame`, `Panel`, `SparseSeries`, and `SparseDataFrame` are deprecated in favor of using `.iat[]` or `.at[]` accessors (GH15269).

• Passing a non-existent column in `.to_excel(..., columns=)` is deprecated and will raise a `KeyError` in the future (GH17295).

• `raise_on_error` parameter to `Series.where()`, `Series.mask()`, `DataFrame.where()`, `DataFrame.mask()` is deprecated, in favor of `errors=` (GH14968).

• Using `DataFrame.rename_axis()` and `Series.rename_axis()` to alter index or column `labels` is now deprecated in favor of using `.rename`. `rename_axis` may still be used to alter the name of the index or columns (GH17833).

• `reindex_axis()` has been deprecated in favor of `reindex()`. See here for more (GH17833).

### Series.select and DataFrame.select

The Series.select() and DataFrame.select() methods are deprecated in favor of using `df.loc[labels.map(crit)]` (GH12401).

```python
In [72]: df = pd.DataFrame({'A': [1, 2, 3], 'index=['foo', 'bar', 'baz']})

In [3]: df.select(lambda x: x in ['bar', 'baz'])
FutureWarning: select is deprecated and will be removed in a future release. You can use ... .loc[crit] as a replacement
Out[3]:
   A
bar 2
baz 3

In [73]: df.loc[df.index.map(lambda x: x in ['bar', 'baz'])]
Out[73]:
   A
bar 2
baz 3
[2 rows x 1 columns]
```
**Series.argmax and Series.argmin**

The behavior of `Series.argmax()` and `Series.argmin()` have been deprecated in favor of `Series.idxmax()` and `Series.idxmin()`, respectively (GH16830).

For compatibility with NumPy arrays, `pd.Series` implements `argmax` and `argmin`. Since pandas 0.13.0, `argmax` has been an alias for `pandas.Series.idxmax()`, and `argmin` has been an alias for `pandas.Series.idxmin()`. They return the label of the maximum or minimum, rather than the position.

We’ve deprecated the current behavior of `Series.argmax` and `Series.argmin`. Using either of these will emit a `FutureWarning`. Use `Series.idxmax()` if you want the label of the maximum. Use `Series.values.argmax()` if you want the position of the maximum. Likewise for the minimum. In a future release `Series.argmax` and `Series.argmin` will return the position of the maximum or minimum.

**Removal of prior version deprecations/changes**

- `read_excel()` has dropped the `has_index_names` parameter (GH10967)
- The `pd.options.display.height` configuration has been dropped (GH3663)
- The `pd.options.display.line_width` configuration has been dropped (GH2881)
- The `pd.options.display.mpl_style` configuration has been dropped (GH12190)
- Index has dropped the `.sym_diff()` method in favor of `.symmetric_difference()` (GH12591)
- Categorical has dropped the `.order()` and `.sort()` methods in favor of `.sort_values()` (GH12882)
- `eval()` and `DataFrame.eval()` have changed the default of `inplace` from `None` to `False` (GH11149)
- The function `get_offset_name` has been dropped in favor of the `.freqstr` attribute for an offset (GH11834)
- pandas no longer tests for compatibility with hdf5-files created with pandas < 0.11 (GH17404).

**Performance improvements**

- Improved performance of instantiating `SparseDataFrame` (GH16773)
- `Series.dt` no longer performs frequency inference, yielding a large speedup when accessing the attribute (GH17210)
- Improved performance of `set_categories()` by not materializing the values (GH17508)
- `Timestamp.microsecond` no longer re-computes on attribute access (GH17331)
- Improved performance of the `CategoricalIndex` for data that is already categorical dtype (GH17513)
- Improved performance of `RangeIndex.min()` and `RangeIndex.max()` by using `RangeIndex` properties to perform the computations (GH17607)
Documentation changes

- Several NaT method docstrings (e.g. NaT.ctime()) were incorrect (GH17327)
- The documentation has had references to versions < v0.17 removed and cleaned up (GH17442, GH17442, GH17404 & GH17504)

Bug fixes

Conversion

- Bug in assignment against datetime-like data with int may incorrectly convert to datetime-like (GH14145)
- Bug in assignment against int64 data with np.ndarray with float64 dtype may keep int64 dtype (GH14001)
- Fixed the return type of IntervalIndex.is_non_overlapping_monotonic to be a Python bool for consistency with similar attributes/methods. Previously returned a numpy.bool_ (GH17237)
- Bug in IntervalIndex.is_non_overlapping_monotonic when intervals are closed on both sides and overlap at a point (GH16560)
- Bug in Series.fillna() returns frame when inplace=True and value is dict (GH16156)
- Bug in Timestamp.weekday_name returning a UTC-based weekday name when localized to a timezone (GH17354)
- Bug in Timestamp.replace when replacing tzinfo around DST changes (GH15683)
- Bug in Timedelta construction and arithmetic that would not propagate the Overflow exception (GH17367)
- Bug in astype() converting to object dtype when passed extension type classes (DatetimeTZDtype, CategoricalDtype) rather than instances. Now a TypeError is raised when a class is passed (GH17780).
- Bug in to_numeric() in which elements were not always being coerced to numeric when errors='coerce' (GH17007, GH17125)
- Bug in DataFrame and Series constructors where range objects are converted to int32 dtype on Windows instead of int64 (GH16804)

Indexing

- When called with a null slice (e.g. df.iloc[:]), the .iloc and .loc indexers return a shallow copy of the original object. Previously they returned the original object. (GH13873).
- When called on an unsorted MultiIndex, the loc indexer now will raise UnsortedIndexError only if proper slicing is used on non-sorted levels (GH16734).
- Fixes regression in 0.20.3 when indexing with a string on a TimedeltaIndex (GH16896).
- Fixed TimedeltaIndex.get_loc() handling of np.timedelat64 inputs (GH16909).
- Fix MultiIndex.sort_index() ordering when ascending argument is a list, but not all levels are specified, or are in a different order (GH16934).
- Fixes bug where indexing with np.inf caused an OverflowError to be raised (GH16957)
- Bug in reindexing on an empty CategoricalIndex (GH16770)
- Fixes DataFrame.loc for setting with alignment and tz-aware DatetimeIndex (GH16889)
- Avoids IndexError when passing an Index or Series to .iloc with older numpy (GH17193)
• Allow unicode empty strings as placeholders in multilevel columns in Python 2 (GH17099)
• Bug in .iloc when used with inplace addition or assignment and an int indexer on a MultiIndex causing the wrong indexes to be read from and written to (GH17148)
• Bug in .isin() in which checking membership in empty Series objects raised an error (GH16991)
• Bug in CategoricalIndex reindexing in which specified indices containing duplicates were not being respected (GH17323)
• Bug in intersection of RangeIndex with negative step (GH17296)
• Bug in IntervalIndex where performing a scalar lookup fails for included right endpoints of non-overlapping monotonic decreasing indexes (GH16417, GH17271)
• Bug in DataFrame.first_valid_index() and DataFrame.last_valid_index() when no valid entry (GH17400)
• Bug in Series.rename() when called with a callable, incorrectly alters the name of the Series, rather than the name of the Index. (GH17407)
• Bug in String.str_get() raises IndexError instead of inserting NaNs when using a negative index. (GH17704)

IO

• Bug in read_hdf() when reading a timezone aware index from fixed format HDFStore (GH17618)
• Bug in read_csv() in which columns were not being thoroughly de-duplicated (GH17060)
• Bug in read_csv() in which specified column names were not being thoroughly de-duplicated (GH17095)
• Bug in read_csv() in which non integer values for the header argument generated an unhelpful / unrelated error message (GH16338)
• Bug in read_csv() in which memory management issues in exception handling, under certain conditions, would cause the interpreter to segfault (GH14696, GH16798).
• Bug in read_csv() when called with low_memory=False in which a CSV with at least one column > 2GB in size would incorrectly raise a MemoryError (GH16798).
• Bug in read_csv() when called with a single-element list header would return a DataFrame of all NaN values (GH7757)
• Bug in DataFrame.to_csv() defaulting to ‘ascii’ encoding in Python 3, instead of ‘utf-8’ (GH17097)
• Bug in read_stata() where value labels could not be read when using an iterator (GH16923)
• Bug in read_stata() where the index was not set (GH16342)
• Bug in read_html() where import check fails when run in multiple threads (GH16928)
• Bug in read_csv() where automatic delimiter detection caused a TypeError to be thrown when a bad line was encountered rather than the correct error message (GH13374)
• Bug in DataFrame.to_html() with notebook=True where DataFrames with named indices or non-MultiIndex indices had undesired horizontal or vertical alignment for column or row labels, respectively (GH16792)
• Bug in DataFrame.to_html() in which there was no validation of the justify parameter (GH17527)
• Bug in HDFStore.select() when reading a contiguous mixed-data table featuring VLArray (GH17021)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in \texttt{to\_json()} where several conditions (including objects with unprintable symbols, objects with deep recursion, overlong labels) caused segfaults instead of raising the appropriate exception (GH14256)

\textbf{Plotting}

- Bug in plotting methods using \texttt{secondary\_y} and \texttt{fontsize} not setting secondary axis font size (GH12565)
- Bug when plotting \texttt{timedelta} and \texttt{datetime} dtypes on y-axis (GH16953)
- Line plots no longer assume monotonic x data when calculating x\texttt{lims}, they show the entire lines now even for unsorted x data. (GH11310, GH11471)
- With matplotlib 2.0.0 and above, calculation of x limits for line plots is left to matplotlib, so that its new default settings are applied. (GH15495)
- Bug in \texttt{Series.plot.bar} or \texttt{DataFrame.plot.bar} with \texttt{y} not respecting user-passed \texttt{color} (GH16822)
- Bug causing \texttt{plotting.parallel\_coordinates} to reset the random seed when using random colors (GH17525)

\textbf{GroupBy/resample/rolling}

- Bug in \texttt{DataFrame.resample(...).size()} where an empty \texttt{DataFrame} did not return a \texttt{Series} (GH14962)
- Bug in \texttt{infer\_freq()} causing indices with 2-day gaps during the working week to be wrongly inferred as business daily (GH16624)
- Bug in \texttt{.rolling(...).quantile()} which incorrectly used different defaults than \texttt{Series.quantile()} and \texttt{DataFrame.quantile()} (GH9413, GH16211)
- Bug in \texttt{groupby.transform()} that would coerce boolean dtypes back to float (GH16875)
- Bug in \texttt{Series.resample(...).apply()} where an empty \texttt{Series} modified the source index and did not return the name of a \texttt{Series} (GH14313)
- Bug in \texttt{.rolling(...).apply(...)} with a \texttt{DataFrame} with a DatetimeIndex, a window of a timedelta-convertible and \texttt{min\_periods} >= 1 (GH15305)
- Bug in \texttt{DataFrame.groupby} where index and column keys were not recognized correctly when the number of keys equaled the number of elements on the groupby axis (GH16859)
- Bug in \texttt{groupby.nunique()} with \texttt{TimeGrouper} which cannot handle \texttt{NaT} correctly (GH17575)
- Bug in \texttt{DataFrame.groupby} where a single level selection from a \texttt{MultiIndex} unexpectedly sorts (GH17537)
- Bug in \texttt{DataFrame.groupby} where spurious warning is raised when \texttt{Grouper} object is used to override ambiguous column name (GH17383)
- Bug in \texttt{TimeGrouper} differs when passes as a list and as a scalar (GH17530)
Sparse

- Bug in SparseSeries raises AttributeError when a dictionary is passed in as data (GH16905)
- Bug in SparseDataFrame.fillna() not filling all NaNs when frame was instantiated from SciPy sparse matrix (GH16112)
- Bug in SparseSeries.unstack() and SparseDataFrame.stack() (GH16614, GH15045)
- Bug in make_sparse() treating two numeric/boolean data, which have same bits, as same when array dtype is object (GH17574)
- SparseArray.all() and SparseArray.any() are now implemented to handle SparseArray, these were used but not implemented (GH17570)

Reshaping

- Joining/Merging with a non unique PeriodIndex raised a TypeError (GH16871)
- Bug in crosstab() where non-aligned series of integers were casted to float (GH17005)
- Bug in merging with categorical dtypes with datetimelikes incorrectly raised a TypeError (GH16900)
- Bug when using isin() on a large object series and large comparison array (GH16012)
- Fixes regression from 0.20, Series.aggregate() and DataFrame.aggregate() allow dictionaries as return values again (GH16741)
- Fixes dtype of result with integer dtype input, from pivot_table() when called with margins=True (GH17013)
- Bug in crosstab() where passing two Series with the same name raised a KeyError (GH13279)
- Series.argmin(), Series.argmax(), and their counterparts on DataFrame and groupby objects work correctly with floating point data that contains infinite values (GH13595).
- Bug in unique() where checking a tuple of strings raised a TypeError (GH17108)
- Bug in concat() where order of result index was unpredictable if it contained non-comparable elements (GH17344)
- Fixes regression when sorting by multiple columns on a datetime64 dtype Series with NaT values (GH16836)
- Bug in pivot_table() where the result’s columns did not preserve the categorical dtype of columns when dropna was False (GH17842)
- Bug in DataFrame.drop_duplicates where dropping with non-unique column names raised a ValueError (GH17836)
- Bug in unstack() which, when called on a list of levels, would discard the fillna argument (GH13971)
- Bug in the alignment of range objects and other list-likes with DataFrame leading to operations being performed row-wise instead of column-wise (GH17901)
### Numeric

- Bug in `.clip()` with axis=1 and a list-like for threshold is passed; previously this raised `ValueError` (GH15390)
- `Series.clip()` and `DataFrame.clip()` now treat NA values for upper and lower arguments as `None` instead of raising `ValueError` (GH17276).

### Categorical

- Bug in `Series.isin()` when called with a categorical (GH16639)
- Bug in the categorical constructor with empty values and categories causing the `.categories` to be an empty `Float64Index` rather than an empty Index with object dtype (GH17248)
- Bug in categorical operations with `Series.cat` not preserving the original Series’ name (GH17509)
- Bug in `DataFrame.merge()` failing for categorical columns with boolean/int data types (GH17187)
- Bug in constructing a Categorical/CategoricalDtype when the specified categories are of categorical type (GH17884).

### PyPy

- Compatibility with PyPy in `read_csv()` with `usecols=[<unsorted ints>]` and `read_json()` (GH17351)
- Split tests into cases for CPython and PyPy where needed, which highlights the fragility of index matching with `float('nan'), np.nan` and `NAT` (GH17351)
- Fix `DataFrame.memory_usage()` to support PyPy. Objects on PyPy do not have a fixed size, so an approximation is used instead (GH17228)

### Other

- Bug where some inplace operators were not being wrapped and produced a copy when invoked (GH12962)
- Bug in `eval()` where the `inplace` parameter was being incorrectly handled (GH16732)

### Contributors

A total of 206 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- 3553x +
- Aaron Barber +
- Adam Gleave +
- Adam Smith +
- AdamShamlian +
- Adrian Liaw +
- Alan Velasco +
- Alan Yee +
• Alex B +
• Alex Lubbock +
• Alex Marchenko +
• Alex Rychyk +
• Amol K +
• Andreas Winkler
• Andrew +
• Andrew
• André Jonasson +
• Becky Sweger
• Berkay +
• Bob Haffner +
• Bran Yang
• Brian Tu +
• Brock Mendel +
• Carol Willing +
• Carter Green +
• Chankey Pathak +
• Chris
• Chris Billington
• Chris Filo Gorgolewski +
• Chris Kerr
• Chris M +
• Chris Mazzullo +
• Christian Prinoth
• Christian Stade-Schuldt
• Christoph Moehl +
• DSM
• Daniel Chen +
• Daniel Grady
• Daniel Himmelstein
• Dave Willmer
• David Cook
• David Gwynne
• David Read +
• Dillon Niederhut +
- Douglas Rudd
- Eric Stein +
- Eric Wieser +
- Erik Fredriksen
- Florian Wilhelm +
- Floris Kint +
- Forbidden Donut
- Gabe F +
- Giftlin +
- Giftlin Rajaiah +
- Giulio Pepe +
- Guilherme Beltramini
- Guillem Borrell +
- Hanmin Qin +
- Hendrik Makait +
- Hugues Valois
- Hussain Tamboli +
- Iva Miholic +
- Jan Novotný +
- Jan Rudolph
- Jean Helie +
- Jean-Baptiste Schiratti +
- Jean-Mathieu Deschenes
- Jeff Knupp +
- Jeff Reback
- Jeff Tratner
- JennaVergeynst
- JimSteams206
- Joel Nothman
- John W. O’Brien
- Jon Crall +
- Jon Mease
- Jonathan J. Helmus +
- Joris Van den Bossche
- JosephWagner
- Juarez Bochi
• Julian Kuhlmann +
• Karel De Brabandere
• Kassandra Keeton +
• Keiron Pizzey +
• Keith Webber
• Kerne
• Kevin Sheppard
• Kirk Hansen +
• Licht Takeuchi +
• Lucas Kushner +
• Mahdi Ben Jelloul +
• Makarov Andrey +
• Malgorzata Turzanska +
• Marc Garcia +
• Margaret Sy +
• MarsGuy +
• Matt Bark +
• Matthew Roeschke
• Matti Picus
• Mehmet Ali “Mali” Akmanalp
• Michael Gasvoda +
• Michael Penkov +
• Milo +
• Morgan Stuart +
• Morgan243 +
• Nathan Ford +
• Nick Eubank
• Nick Garvey +
• Oleg Shteynbuk +
• P-Tillmann +
• Pankaj Pandey
• Patrick Luo
• Patrick O’Melveny
• Paul Reidy +
• Paula +
• Peter Quackenbush
• Peter Yanovich +
• Phillip Cloud
• Pierre Haessig
• Pietro Battiston
• Pradyumna Reddy Chinthala
• Prasanjit Prakash
• RobinFiveWords
• Ryan Hendrickson
• Sam Foo
• Sangwoong Yoon +
• Simon Gibbons +
• SimonBaron
• Steven Cutting +
• Sudeep +
• Sylvia +
• T N +
• Telt
• Thomas A Caswell
• Tim Swast +
• Tom Augspurger
• Tong SHEN
• Tuan +
• Utkarsh Upadhyay +
• Vincent La +
• Vivek +
• WANG Aiyong
• WBare
• Wes McKinney
• XF +
• Yi Liu +
• Yosuke Nakabayashi +
• aaron315 +
• ababar4gh +
• aernlund +
• agustín méndez +
• andymaheshw +
• ante328 +
• aviolv +
• bpraggastis
• cbertinato +
• cclauss +
• chernick
• chris-b1
• dkamm +
• dwkenefick
• economy
• faic +
• fding253 +
• gfyung
• guygoldberg +
• hhuuggoo +
• huashuai +
• ian
• iulia +
• jaredsnyder
• jbrockmendel +
• jdeschenes
• jebob +
• jschendel +
• keitakurita
• kernc +
• kiwirob +
• kjford
• linebp
• llloydkirk
• louispotok +
• majiang +
• manikbhandari +
• matthiashuschle +
• mattip
• maxwasserman +
• mjlove12 +
5.11 Version 0.20

5.11.1 Version 0.20.3 (July 7, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

What’s new in v0.20.3

- **Bug fixes**
  - Conversion
  - Indexing
  - IO
  - Plotting
  - Reshaping
  - Categorical
- Contributors
Bug fixes

- Fixed a bug in failing to compute rolling computations of a column-MultiIndexed DataFrame (GH16789, GH16825)
- Fixed a pytest marker failing downstream packages’ tests suites (GH16680)

Conversion

- Bug in pickle compat prior to the v0.20.x series, when UTC is a timezone in a Series/DataFrame/Index (GH16608)
- Bug in Series construction when passing a Series with dtype='category' (GH16524).
- Bug in DataFrame.astype() when passing a Series as the dtype kwarg. (GH16717).

Indexing

- Bug in Float64Index causing an empty array instead of None to be returned from .get(np.nan) on a Series whose index did not contain any NaNs (GH8569)
- Bug in MultiIndex.isin causing an error when passing an empty iterable (GH16777)
- Fixed a bug in a slicing DataFrame/Series that have a TimedeltaIndex (GH16637)

IO

- Bug in read_csv() in which files weren’t opened as binary files by the C engine on Windows, causing EOF characters mid-field, which would fail (GH16039, GH16559, GH16675)
- Bug in read_hdf() in which reading a Series saved to an HDF file in ‘fixed’ format fails when an explicit mode='r' argument is supplied (GH16583)
- Bug in DataFrame.to_latex() where bold_rows was wrongly specified to be True by default, whereas in reality row labels remained non-bold whatever parameter provided. (GH16707)
- Fixed an issue with DataFrame.style() where generated element ids were not unique (GH16780)
- Fixed loading a DataFrame with a PeriodIndex, from a format='fixed' HDFStore, in Python 3, that was written in Python 2 (GH16781)

Plotting

- Fixed regression that prevented RGB and RGBA tuples from being used as color arguments (GH16233)
- Fixed an issue with DataFrame.plot.scatter() that incorrectly raised a KeyError when categorical data is used for plotting (GH16199)
Reshaping

- PeriodIndex / TimedeltaIndex.join was missing the sort= kwarg (GH16541)
- Bug in joining on a MultiIndex with a category dtype for a level (GH16627).
- Bug in merge() when merging/joining with multiple categorical columns (GH16767)

Categorical

- Bug in DataFrame.sort_values not respecting the kind parameter with categorical data (GH16793)

Contributors

A total of 20 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Bran Yang
- Chris
- Chris Kerr +
- DSM
- David Gwynne
- Douglas Rudd
- Forbidden Donut +
- Jeff Reback
- Joris Van den Bossche
- Karel De Brabandere +
- Peter Quackenbush +
- Pradyumna Reddy Chinthala +
- Telt +
- Tom Augspurger
- chris-b1
- gfyoun
- ian +
- jdeschenes +
- kjford +
- ri938 +
5.11.2 Version 0.20.2 (June 4, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

What’s new in v0.20.2

- **Enhancements**
- **Performance improvements**
- **Bug fixes**
  - Conversion
  - Indexing
  - IO
  - Plotting
  - GroupBy/resample/rolling
  - Sparse
  - Reshaping
  - Numeric
  - Categorical
  - Other
- **Contributors**

**Enhancements**


- Series provides a `to_latex` method (GH16180)

- A new groupby method `ngroup()`, parallel to the existing `cumcount()`, has been added to return the group order (GH11642); see [here](#).

**Performance improvements**

- Performance regression fix when indexing with a list-like (GH16285)

- Performance regression fix for MultiIndexes (GH16319, GH16346)

- Improved performance of `.clip()` with scalar arguments (GH15400)

- Improved performance of groupby with categorical groupers (GH16413)

- Improved performance of `MultiIndex.remove_unused_levels()` (GH16556)
Bug fixes

- Silenced a warning on some Windows environments about “tput: terminal attributes: No such device or address” when detecting the terminal size. This fix only applies to python 3 (GH16496)
- Bug in using pathlib.Path or py.path.local objects with io functions (GH16291)
- Bug in Index.symmetric_difference() on two equal MultiIndex’s, results in a TypeError (GH13490)
- Bug in DataFrame.update() with overwrite=False and NaN values (GH15593)
- Passing an invalid engine to read_csv() now raises an informative ValueError rather than UnboundLocalError. (GH16511)
- Bug in unique() on an array of tuples (GH16519)
- Bug in cut() when labels are set, resulting in incorrect label ordering (GH16459)
- Fixed a compatibility issue with IPython 6.0’s tab completion showing deprecation warnings on Categoricals (GH16409)

Conversion

- Bug in to_numeric() in which empty data inputs were causing a segfault of the interpreter (GH16302)
- Silence numpy warnings when broadcasting DataFrame to Series with comparison ops (GH16378, GH16306)

Indexing

- Bug in DataFrame.reset_index(level=) with single level index (GH16263)
- Bug in partial string indexing with a monotonic, but not strictly-monotonic, index incorrectly reversing the slice bounds (GH16515)
- Bug in MultiIndex.remove_unused_levels() that would not return a MultiIndex equal to the original. (GH16556)

IO

- Bug in read_csv() when comment is passed in a space delimited text file (GH16472)
- Bug in read_csv() not raising an exception with nonexistent columns in usecols when it had the correct length (GH14671)
- Bug that would force importing of the clipboard routines unnecessarily, potentially causing an import error on startup (GH16288)
- Bug that raised IndexError when HTML-rendering an empty DataFrame (GH15953)
- Bug in read_csv() in which tarfile object inputs were raising an error in Python 2.x for the C engine (GH16530)
- Bug where DataFrame.to_html() ignored the index_names parameter (GH16493)
- Bug where pd.read_hdf() returns numpy strings for index names (GH13492)
- Bug in HDFStore.select_as_multiple() where start/stop arguments were not respected (GH16209)
Plotting

- Bug in DataFrame.plot with a single column and a list-like color (GH3486)
- Bug in plot where NaT in DatetimeIndex results in Timestamp.min (GH12405)
- Bug in DataFrame.boxplot where figsize keyword was not respected for non-grouped boxplots (GH11959)

GroupBy/resample/rolling

- Bug in creating a time-based rolling window on an empty DataFrame (GH15819)
- Bug in rolling.cov() with offset window (GH16058)
- Bug in .resample() and .groupby() when aggregating on integers (GH16361)

Sparse

- Bug in construction of SparseDataFrame from scipy.sparse.dok_matrix (GH16179)

Reshaping

- Bug in DataFrame.stack with unsorted levels in MultiIndex columns (GH16323)
- Bug in pd.wide_to_long() where no error was raised when i was not a unique identifier (GH16382)
- Bug in Series.isin(..) with a list of tuples (GH16394)
- Bug in construction of a DataFrame with mixed dtypes including an all-NaT column. (GH16395)
- Bug in DataFrame.agg() and Series.agg() with aggregating on non-callable attributes (GH16405)

Numeric

- Bug in .interpolate(), where limit_direction was not respected when limit=None (default) was passed (GH16282)

Categorical

- Fixed comparison operations considering the order of the categories when both categoricals are unordered (GH16014)

Other

- Bug in DataFrame.drop() with an empty-list with non-unique indices (GH16270)
Contributors

A total of 34 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Aaron Barber +
• Andrew +
• Becky Sweger +
• Christian Prinoth +
• Christian Stade-Schuldt +
• DSM
• Erik Fredriksen +
• Hugues Valois +
• Jeff Reback
• Jeff Tratner
• JimStearns206 +
• John W. O’Brien
• Joris Van den Bossche
• JosephWagner +
• Keith Webber +
• Mehmet Ali “Mali” Akmanalp +
• Pankaj Pandey
• Patrick Luo +
• Patrick O’Melveny +
• Pietro Battiston
• RobinFiveWords +
• Ryan Hendrickson +
• SimonBaron +
• Tom Augspurger
• WBare +
• bpraggastis +
• chernrick +
• chris-b1
• economy +
• gfyong
• jaredsnyder +
• keitakurita +
• linebp
5.11.3 Version 0.20.1 (May 5, 2017)

This is a major release from 0.19.2 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- New `.agg()` API for Series/DataFrame similar to the groupby-rolling-resample API’s, see here
- Integration with the feather-format, including a new top-level `pd.read_feather()` and DataFrame `.to_feather()` method, see here.
- The `.ix` indexer has been deprecated, see here
- Panel has been deprecated, see here
- Addition of an IntervalIndex and Interval scalar type, see here
- Improved user API when grouping by index levels in `.groupby()`, see here
- Improved support for UInt64 dtypes, see here
- A new orient for JSON serialization, `orient='table'`, that uses the Table Schema spec and that gives the possibility for a more interactive repr in the Jupyter Notebook, see here
- Experimental support for exporting styled DataFrames (DataFrame.style) to Excel, see here
- Window binary corr/cov operations now return a MultiIndexed DataFrame rather than a Panel, as Panel is now deprecated, see here
- Support for S3 handling now uses s3fs, see here
- Google BigQuery support now uses the pandas-gbq library, see here

**Warning:** pandas has changed the internal structure and layout of the code base. This can affect imports that are not from the top-level `pandas.*` namespace, please see the changes here.

Check the API Changes and deprecations before updating.

**Note:** This is a combined release for 0.20.0 and 0.20.1. Version 0.20.1 contains one additional change for backwards-compatibility with downstream projects using pandas’ utils routines. (GH16250)

### What’s new in v0.20.0

- **New features**
  - Method `agg API for DataFrame/Series`
  - Keyword argument `dtype` for data IO
  - Method `.to_datetime()` has gained an `origin` parameter
  - GroupBy enhancements
  - Better support for compressed URLs in `read_csv`
- Pickle file IO now supports compression
- UInt64 support improved
- GroupBy on categoricals
- Table schema output
- SciPy sparse matrix from/to SparseDataFrame
- Excel output for styled DataFrames
- IntervalIndex
- Other enhancements

- Backwards incompatible API changes
  - Possible incompatibility for HDF5 formats created with pandas < 0.13.0
  - Map on Index types now return other Index types
  - Accessing datetime fields of Index now return Index
  - pd.unique will now be consistent with extension types
  - S3 file handling
  - Partial string indexing changes
  - Concat of different float dtypes will not automatically upcast
  - pandas Google BigQuery support has moved
  - Memory usage for Index is more accurate
  - DataFrame.sort_index changes
  - GroupBy describe formatting
  - Window binary corr/cov operations return a MultiIndex DataFrame
  - HDFStore where string comparison
  - Index.intersection and inner join now preserve the order of the left Index
  - Pivot table always returns a DataFrame
  - Other API changes

- Reorganization of the library: privacy changes
  - Modules privacy has changed
  - pandas.errors
  - pandas.testing
  - pandas.plotting
  - Other development changes

- Deprecations
  - Deprecate .ix
  - Deprecate Panel
  - Deprecate groupby.agg() with a dictionary when renaming
- Deprecate .plotting
- Other deprecations
  • Removal of prior version deprecations/changes
  • Performance improvements
  • Bug fixes
    - Conversion
    - Indexing
    - IO
    - Plotting
    - GroupBy/resample/rolling
    - Sparse
    - Reshaping
    - Numeric
    - Other
- Contributors

New features

Method agg API for DataFrame/Series

Series & DataFrame have been enhanced to support the aggregation API. This is a familiar API from groupby, window operations, and resampling. This allows aggregation operations in a concise way by using `agg()` and `transform()`. The full documentation is here (GH1623).

Here is a sample

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
                       index=pd.date_range('1/1/2000', periods=10))
...:

In [2]: df.iloc[3:7] = np.nan

In [3]: df
```

```
Out[3]:
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-01</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-1.135632</td>
<td>1.212112</td>
<td>-0.173215</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.119209</td>
<td>-1.044236</td>
<td>-0.861849</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-1.039268</td>
<td>-0.370647</td>
<td>-1.157892</td>
</tr>
</tbody>
</table>
```

(continues on next page)
One can operate using string function names, callables, lists, or dictionaries of these.

Using a single function is equivalent to `apply`.

```python
In [4]: df.agg('sum')
Out[4]:
A     -1.068226
B     -1.387015
C     -4.892029
Length: 3, dtype: float64
```

Multiple aggregations with a list of functions.

```python
In [5]: df.agg(['sum', 'min'])
Out[5]:
          A       B        C
sum -1.068226 -1.387015  -4.892029
min -1.135632 -1.478427  -1.715002

[2 rows x 3 columns]
```

Using a dict provides the ability to apply specific aggregations per column. You will get a matrix-like output of all of the aggregators. The output has one column per unique function. Those functions applied to a particular column will be `NaN`:

```python
In [6]: df.agg({'A': ['sum', 'min'], 'B': ['min', 'max']})
Out[6]:
          A       B       C
sum -1.068226 NaN     NaN
min -1.135632 -1.478427  1.212112
max  NaN     NaN     1.212112

[3 rows x 2 columns]
```

The API also supports a `.transform()` function for broadcasting results.

```python
In [7]: df.transform(['abs', lambda x: x - x.min()])
Out[7]:
          A       B       C
    abs <lambda> abs <lambda> abs <lambda>
2000-01-01  0.469112  1.604745  0.282863  1.195563  1.509059  0.205944
2000-01-02  1.135632  0.000000  1.212112  2.690539  0.173215  1.541787
2000-01-03  0.119209  1.254841  1.044236  0.434191  0.861849  0.853153
2000-01-04  NaN       NaN       NaN       NaN       NaN       NaN
2000-01-05  NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06  NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07  NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  0.113648  1.249281  1.478427  0.000000  0.524988  2.239990
2000-01-09  0.404705  1.540338  0.577046  2.055473  1.715002  0.000000
2000-01-10  1.039268  0.096364  0.370647  1.107780  1.157892  0.557110
```

(continues on next page)
When presented with mixed dtypes that cannot be aggregated, `.agg()` will only take the valid aggregations. This is similar to how `groupby .agg()` works. (GH15015)

```python
In [8]: df = pd.DataFrame({'A': [1, 2, 3],
                     ...:                     'B': [1., 2., 3.],
                     ...:                     'C': ['foo', 'bar', 'baz'],
                     ...:                     'D': pd.date_range('20130101', periods=3))

In [9]: df.dtypes
Out[9]:
A     int64
B     float64
C   object
D  datetime64[ns]
Length: 4, dtype: object

In [10]: df.agg(['min', 'sum'])
Out[10]:
      A    B      C     D
min  1.0  1.0  bar  2013-01-01
sum  6.0  6.0 foobarbaz    NaT
[2 rows x 4 columns]
```

**Keyword argument `dtype` for data IO**

The 'python' engine for `read_csv()`, as well as the `read_fwf()` function for parsing fixed-width text files and `read_excel()` for parsing Excel files, now accept the `dtype` keyword argument for specifying the types of specific columns (GH14295). See the io docs for more information.

```python
In [11]: data = "a b\n   2\n3 4"
In [12]: pd.read_fwf(StringIO(data)).dtypes
Out[12]:
a   int64
b   int64
Length: 2, dtype: object

In [13]: pd.read_fwf(StringIO(data), dtype={'a': 'float64', 'b': 'object'}).dtypes
Out[13]:
a   float64
b   object
Length: 2, dtype: object
```
Method `to_datetime()` has gained an origin parameter

to_datetime() has gained a new parameter, `origin`, to define a reference date from where to compute the resulting timestamps when parsing numerical values with a specific `unit` specified. (GH11276, GH11745)

For example, with 1960-01-01 as the starting date:

```python
In [14]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[14]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype='datetime64[ns]', freq=None)
```

The default is set at `origin='unix'`, which defaults to 1970-01-01 00:00:00, which is commonly called ‘unix epoch’ or POSIX time. This was the previous default, so this is a backward compatible change.

```python
In [15]: pd.to_datetime([1, 2, 3], unit='D')
Out[15]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype='datetime64[ns]', freq=None)
```

GroupBy enhancements

Strings passed to `DataFrame.groupby()` as the by parameter may now reference either column names or index level names. Previously, only column names could be referenced. This allows to easily group by a column and index level at the same time. (GH5677)

```python
In [16]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
                ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
In [17]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])
In [18]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
               'B': np.arange(8)},
               index=index)
In [19]: df.groupby(['second', 'A']).sum()
```

(continues on next page)
Better support for compressed URLs in `read_csv`

The compression code was refactored (GH12688). As a result, reading dataframes from URLs in `read_csv()` or `read_table()` now supports additional compression methods: xz, bz2, and zip (GH14570). Previously, only gzip compression was supported. By default, compression of URLs and paths are now inferred using their file extensions. Additionally, support for bz2 compression in the python 2 C-engine improved (GH14874).

```python
In [21]: url = ('https://github.com/{repo}/raw/{branch}/{path}'
        ....: .format(repo='pandas-dev/pandas',
        ....:         branch='main',
        ....:         path='pandas/tests/io/parser/data/salaries.csv.bz2'))

# default, infer compression
In [22]: df = pd.read_csv(url, sep='\t', compression='infer')

# explicitly specify compression
In [23]: df = pd.read_csv(url, sep='\t', compression='bz2')

In [24]: df.head(2)
Out[24]:
   S  X  E  M
0  13876 1  1  1
1  11608 1  3  0

[2 rows x 4 columns]
```

Pickle file IO now supports compression

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can now read from and write to compressed pickle files. Compression methods can be an explicit parameter or be inferred from the file extension. See the docs here.

```python
In [25]: df = pd.DataFrame({'A': np.random.randn(1000),
                     ....: 'B': 'foo',
                     ....: 'C': pd.date_range('20130101', periods=1000, freq='s'))

Using an explicit compression type
```
In [26]: df.to_pickle("data.pkl.compress", compression="gzip")

In [27]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [28]: rt.head()
Out[28]:
          A          B          C
0 -1.344312 foo 2013-01-01 00:00:00
1  0.844885 foo 2013-01-01 00:00:01
2  1.075770 foo 2013-01-01 00:00:02
3 -0.109050 foo 2013-01-01 00:00:03
4  1.643563 foo 2013-01-01 00:00:04

[5 rows x 3 columns]

The default is to infer the compression type from the extension (compression='infer'):

In [29]: df.to_pickle("data.pkl.gz")

In [30]: rt = pd.read_pickle("data.pkl.gz")

In [31]: rt.head()
Out[31]:
          A          B          C
0 -1.344312 foo 2013-01-01 00:00:00
1  0.844885 foo 2013-01-01 00:00:01
2  1.075770 foo 2013-01-01 00:00:02
3 -0.109050 foo 2013-01-01 00:00:03
4  1.643563 foo 2013-01-01 00:00:04

[5 rows x 3 columns]

In [32]: df["A"].to_pickle("s1.pkl.bz2")

In [33]: rt = pd.read_pickle("s1.pkl.bz2")

In [34]: rt.head()
Out[34]:
       0
0 -1.344312
1  0.844885
2  1.075770
3 -0.109050
4  1.643563
Name: A, Length: 5, dtype: float64
**UInt64 support improved**

pandas has significantly improved support for operations involving unsigned, or purely non-negative, integers. Previously, handling these integers would result in improper rounding or data-type casting, leading to incorrect results. Notably, a new numerical index, UInt64Index, has been created (GH14937)

```python
In [1]: idx = pd.UInt64Index([1, 2, 3])
In [2]: df = pd.DataFrame({'A': ['a', 'b', 'c'], index=idx})
In [3]: df.index
Out[3]: UInt64Index([1, 2, 3], dtype='uint64')
```

- Bug in converting object elements of array-like objects to unsigned 64-bit integers (GH4471, GH14982)
- Bug in Series.unique() in which unsigned 64-bit integers were causing overflow (GH14721)
- Bug in DataFrame construction in which unsigned 64-bit integer elements were being converted to objects (GH14881)
- Bug in pd.read_csv() in which unsigned 64-bit integer elements were being improperly converted to the wrong data types (GH14983)
- Bug in pd.unique() in which unsigned 64-bit integers were causing overflow (GH14915)
- Bug in pd.value_counts() in which unsigned 64-bit integers were being erroneously truncated in the output (GH14934)

**GroupBy on categoricals**

In previous versions, `.groupby(..., sort=False)` would fail with a ValueError when grouping on a categorical series with some categories not appearing in the data. (GH13179)

```python
In [35]: chromosomes = np.r_[np.arange(1, 23).astype(str), ['X', 'Y']]

In [36]: df = pd.DataFrame({
    ....:     'A': np.random.randint(100),
    ....:     'B': np.random.randint(100),
    ....:     'C': np.random.randint(100),
    ....:     'chromosomes': pd.Categorical(np.random.choice(chromosomes, 100), categories=chromosomes, ordered=True))

In [37]: df
Out[37]:
     A  B  C        chromosomes
0  87  22  81          4
1  87  22  81         13
2  87  22  81         22
3  87  22  81          2
4  87  22  81          6
... ... ...          ...
95  87  22  81          8
96  87  22  81         11
97  87  22  81          X
98  87  22  81          1
```

(continues on next page)
Previous behavior:

```python
In [3]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
---------------------------------------------------------------------------
ValueError: items in new_categories are not the same as in old categories
```

New behavior:

```python
In [38]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
```

```
<table>
<thead>
<tr>
<th>chromosomes</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>348</td>
<td>88</td>
<td>324</td>
</tr>
<tr>
<td>3</td>
<td>348</td>
<td>88</td>
<td>324</td>
</tr>
<tr>
<td>4</td>
<td>348</td>
<td>88</td>
<td>324</td>
</tr>
<tr>
<td>5</td>
<td>261</td>
<td>66</td>
<td>243</td>
</tr>
<tr>
<td>6</td>
<td>174</td>
<td>44</td>
<td>162</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>348</td>
<td>88</td>
<td>324</td>
</tr>
<tr>
<td>X</td>
<td>348</td>
<td>88</td>
<td>324</td>
</tr>
<tr>
<td>Y</td>
<td>435</td>
<td>110</td>
<td>405</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
[24 rows x 3 columns]
```

**Table schema output**

The new orient 'table' for `DataFrame.to_json()` will generate a Table Schema compatible string representation of the data.

```python
In [39]: df = pd.DataFrame(
   ....:     {'A': [1, 2, 3],
   ....:     'B': ['a', 'b', 'c'],
   ....:     'C': pd.date_range('2016-01-01', freq='d', periods=3),
   ....:     index=pd.Index(range(3), name='idx'))
```

```python
In [40]: df
Out[40]:
```

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a 2016-01-01</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>b 2016-01-02</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>c 2016-01-03</td>
</tr>
</tbody>
</table>
```

```
[3 rows x 3 columns]
```
In [41]: df.to_json(orient='table')
Out[41]: {'schema': {'fields': [{'name': 'idx', 'type': 'integer'}, {'name': 'A', 'type': 'integer'}, {'name': 'B', 'type': 'string'}, {'name': 'C', 'type': 'datetime'}], 'primaryKey': ['idx'], 'pandas_version': '1.4.0'}, 'data': [{'idx': 0, 'A': 1, 'B': 'a', 'C': '2016-01-01T00:00:00.000Z'}, {'idx': 1, 'A': 2, 'B': 'b', 'C': '2016-01-02T00:00:00.000Z'}, {'idx': 2, 'A': 3, 'B': 'c', 'C': '2016-01-03T00:00:00.000Z'}]}

See **IO: Table Schema for more information**.

Additionally, the repr for DataFrame and Series can now publish this JSON Table schema representation of the Series or DataFrame if you are using IPython (or another frontend like nteract using the Jupyter messaging protocol). This gives frontends like the Jupyter notebook and nteract more flexibility in how they display pandas objects, since they have more information about the data. You must enable this by setting the display.html.table_schema option to True.

### SciPy sparse matrix from/to SparseDataFrame

pandas now supports creating sparse dataframes directly from scipy.sparse.spmatrix instances. See the documentation for more information. (GH4343)

All sparse formats are supported, but matrices that are not in **COOrdinate** format will be converted, copying data as needed.

```python
from scipy.sparse import csr_matrix
arr = np.random.random(size=(1000, 5))
arr[arr < .9] = 0
sp_arr = csr_matrix(arr)
sdf = pd.SparseDataFrame(sp_arr)
sdf
```

To convert a SparseDataFrame back to sparse SciPy matrix in COO format, you can use:

```python
sdf.to_coo()
```

### Excel output for styled DataFrames

Experimental support has been added to export DataFrame.style formats to Excel using the openpyxl engine. (GH15530)

For example, after running the following, `styled.xlsx` renders as below:

```python
In [42]: np.random.seed(24)

In [43]: df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
In [44]: df = pd.concat([df, pd.DataFrame(np.random.RandomState(24).randn(10, 4),
                      columns=list('BCDE'))],
               axis=1)
               ...
```

(continues on next page)
In [45]: df.iloc[0, 2] = np.nan

In [46]: df

Out[46]:
    A         B         C         D         E
0  1.0  1.329212 NaN -0.316280 -0.990810
1  2.0 -1.070816 -1.438713  0.564417  0.295722
2  3.0  1.626404  0.219565  0.678805  1.889273
3  4.0  0.961538  0.104011 -0.481165  0.850229
4  5.0  1.453425  1.057737  0.165562  0.515018
5  6.0 -1.336936  0.562861  1.392855 -0.063328
6  7.0  0.121668  1.207603 -0.002040  1.627796
7  8.0  0.354493  1.037528 -0.385684  0.519818
8  9.0  1.686583 -1.325963  1.428984 -2.089354
9 10.0 -0.129820  0.631523 -0.586538  0.290720
[10 rows x 5 columns]

In [47]: styled = (df.style
                .applymap(lambda val: 'color:red;' if val < 0 else 'color:black;')
                .highlight_max())

In [48]: styled.to_excel('styled.xlsx', engine='openpyxl')

See the Style documentation for more detail.
IntervalIndex

pandas has gained an IntervalIndex with its own dtype, interval as well as the Interval scalar type. These allow first-class support for interval notation, specifically as a return type for the categories in `cut()` and `qcut()`. The IntervalIndex allows some unique indexing, see the docs. (GH7640, GH8625)

**Warning:** These indexing behaviors of the IntervalIndex are provisional and may change in a future version of pandas. Feedback on usage is welcome.

Previous behavior:

The returned categories were strings, representing Intervals

```
In [1]: c = pd.cut(range(4), bins=2)
In [2]: c
Out[2]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3], (1.5, 3]]
Categories (2, object): [(-0.003, 1.5] < (1.5, 3]]
```

```
In [3]: c.categories
Out[3]: Index(['(-0.003, 1.5]', '(1.5, 3]'], dtype='object')
```

New behavior:

```
In [49]: c = pd.cut(range(4), bins=2)
In [50]: c
Out[50]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]
```

```
In [51]: c.categories
Out[51]: IntervalIndex([(-0.003, 1.5], (1.5, 3.0]], dtype='interval[float64, right]')
```

Furthermore, this allows one to bin other data with these same bins, with NaN representing a missing value similar to other dtypes.

```
In [52]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[52]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]
```

An IntervalIndex can also be used in Series and DataFrame as the index.

```
In [53]: df = pd.DataFrame({'A': range(4),
                   'B': pd.cut([0, 3, 1, 1], bins=c.categories)}).set_index('B')
In [54]: df
Out[54]:
```

(continues on next page)
Selecting via a specific interval:

```python
In [55]: df.loc[pd.Interval(1.5, 3.0)]
Out[55]:
   A
Name: (1.5, 3.0], Length: 1, dtype: int64
```

Selecting via a scalar value that is contained in the intervals.

```python
In [56]: df.loc[0]
Out[56]:
   A
   B
(-0.003, 1.5]  0
(-0.003, 1.5]  2
(-0.003, 1.5]  3
[3 rows x 1 columns]
```

Other enhancements

- **DataFrame.rolling()** now accepts the parameter `closed='right'|'left'|'both'|'neither'` to choose the rolling window-endpoint closedness. See the documentation (GH13965)
- Integration with the feather-format, including a new top-level `pd.read_feather()` and `DataFrame.to_feather()` method, see [here](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_feather.html).
- **Series.str.replace()** now accepts a callable, as replacement, which is passed to `re.sub` (GH15055)
- **Series.str.replace()** now accepts a compiled regular expression as a pattern (GH15446)
- **Series.sort_index** accepts parameters `kind` and `na_position` (GH13589, GH14444)
- **DataFrame and DataFrame.groupby()** have gained a `nunique()` method to count the distinct values over an axis (GH14336, GH15197).
- **DataFrame** has gained a `melt()` method, equivalent to `pd.melt()`, for unpivoting from a wide to long format (GH12640).
- **pd.read_excel()** now preserves sheet order when using `sheetname=None` (GH9930)
- Multiple offset aliases with decimal points are now supported (e.g. `0.5min` is parsed as `30s`) (GH8419)
- **.isnull()** and **.notnull()** have been added to **Index** object to make them more consistent with the **Series** API (GH15300)
- New **UnsortedIndexError** (subclass of **KeyError**) raised when indexing/slicing into an unsorted MultiIndex (GH11897). This allows differentiation between errors due to lack of sorting or an incorrect key. See [here](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.MultiIndex.sort_values.html)
• MultiIndex has gained a `.to_frame()` method to convert to a DataFrame (GH12397)
• `pd.cut` and `pd.qcut` now support datetime64 and timedelta64 dtypes (GH14714, GH14798)
• `pd.qcut` has gained the duplicates='raise' | 'drop' option to control whether to raise on duplicated edges (GH7751)
• Series provides a `to_excel` method to output Excel files (GH8825)
• The `usecols` argument in `pd.read_csv()` now accepts a callable function as a value (GH14154)
• The skiprows argument in `pd.read_csv()` now accepts a callable function as a value (GH10882)
• The nrows and chunksize arguments in `pd.read_csv()` are supported if both are passed (GH6774, GH15755)
• `DataFrame.plot` now prints a title above each subplot if `subplots=True` and `title` is a list of strings (GH14753)
• `DataFrame.plot` can pass the matplotlib 2.0 default color cycle as a single string as color parameter, see here. (GH15516)
• `Series.interpolate()` now supports timedelta as an index type with method='time' (GH6424)
• Addition of a level keyword to `DataFrame/Series.rename` to rename labels in the specified level of a MultiIndex (GH4160).
• `DataFrame.reset_index()` will now interpret a tuple `index.name` as a key spanning across levels of columns, if this is a MultiIndex (GH16164)
• `Timedelta.isoformat` method added for formatting Timedeltas as an ISO 8601 duration. See the `Timedelta docs` (GH15136)
• `.select_dtypes()` now allows the string `datetimetz` to generically select datetimes with tz (GH14910)
• The `.to_latex()` method will now accept multicol and multirow arguments to use the accompanying LaTeX enhancements
• `pd.merge_asof()` gained the option direction='backward' | 'forward' | 'nearest' (GH14887)
• `Series/DataFrame.asfreq()` have gained a fill_value parameter, to fill missing values (GH3715).
• `Series/DataFrame.resample.asfreq` have gained a fill_value parameter, to fill missing values during resampling (GH3715).
• `pandas.util.hash_pandas_object()` has gained the ability to hash a MultiIndex (GH15224)
• `Series/DataFrame.squeeze()` have gained the axis parameter. (GH15339)
• `DataFrame.to_excel()` has a new freeze_panes parameter to turn on Freeze Panes when exporting to Excel (GH15160)
• `pd.read_html()` will parse multiple header rows, creating a MultiIndex header. (GH13434).
• HTML table output skips colspan or rowspan attribute if equal to 1. (GH15403)
• `pandas.ioformats.style.Styler` template now has blocks for easier extension, see the example notebook (GH15649)
• `Styler.render()` now accepts **kwargs to allow user-defined variables in the template (GH15649)
• Compatibility with Jupyter notebook 5.0; MultiIndex column labels are left-aligned and MultiIndex row-labels are top-aligned (GH15379)
• `TimedeltaIndex` now has a custom date-tick formatter specifically designed for nanosecond level precision (GH8711)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- pd.api.types.union_categoricals gained the ignore_ordered argument to allow ignoring the ordered attribute of unioned categoricals (GH13410). See the categorical union docs for more information.
- DataFrame.to_latex() and DataFrame.to_string() now allow optional header aliases. (GH15536)
- Re-enable the parse_dates keyword of pd.read_excel() to parse string columns as dates (GH14326)
- Added .empty property to subclasses of Index. (GH15270)
- Enabled floor division for Timedelta and TimedeltaIndex (GH15828)
- pandas.io.json.json_normalize() gained the option errors='ignore' | 'raise'; the default is errors='raise' which is backward compatible. (GH14583)
- pandas.io.json.json_normalize() with an empty list will return an empty DataFrame (GH15534)
- pandas.io.json.json_normalize() has gained a sep option that accepts str to separate joined fields; the default is ".", which is backward compatible. (GH14883)
- MultiIndex.remove_unused_levels() has been added to facilitate removing unused levels. (GH15694)
- pd.read_csv() will now raise a ParserError error whenever any parsing error occurs (GH15913, GH15925)
- pd.read_csv() now supports the error_bad_lines and warn_bad_lines arguments for the Python parser (GH15925)
- The display.show_dimensions option can now also be used to specify whether the length of a Series should be shown in its repr (GH7117).
- parallel_coordinates() has gained a sort_labels keyword argument that sorts class labels and the colors assigned to them (GH15908)
- Options added to allow one to turn on/off using bottleneck and numexpr, see here (GH16157)
- DataFrame.style.bar() now accepts two more options to further customize the bar chart. Bar alignment is set with align='left' | 'mid' | 'zero', the default is "left", which is backward compatible; You can now pass a list of color=[color_negative, color_positive]. (GH14757)

Backwards incompatible API changes

Possible incompatibility for HDF5 formats created with pandas < 0.13.0

pd.TimeSeries was deprecated officially in 0.17.0, though has already been an alias since 0.13.0. It has been dropped in favor of pd.Series. (GH15098).

This may cause HDF5 files that were created in prior versions to become unreadable if pd.TimeSeries was used. This is most likely to be for pandas < 0.13.0. If you find yourself in this situation. You can use a recent prior version of pandas to read in your HDF5 files, then write them out again after applying the procedure below.

In [2]: s = pd.TimeSeries([1, 2, 3], index=pd.date_range('20130101', periods=3))
In [3]: s
Out[3]:
2013-01-01  1
2013-01-02  2
2013-01-03  3
Freq: D, dtype: int64
In [4]: type(s)
Out[4]: pandas.core.series.TimeSeries
In [5]: s = pd.Series(s)

In [6]: s
Out[6]:
2013-01-01  1
2013-01-02  2
2013-01-03  3
Freq: D, dtype: int64

In [7]: type(s)
Out[7]: pandas.core.series.Series

Map on Index types now return other Index types

map on an Index now returns an Index, not a numpy array (GH12766)

In [57]: idx = pd.Index([1, 2])

In [58]: idx
Out[58]: Int64Index([1, 2], dtype='int64')

In [59]: mi = pd.MultiIndex.from_tuples([(1, 2), (2, 4)])

In [60]: mi
Out[60]: MultiIndex([(1, 2),
(2, 4)],
)

Previous behavior:

In [5]: idx.map(lambda x: x * 2)
Out[5]: array([2, 4])

In [6]: idx.map(lambda x: (x, x * 2))
Out[6]: array([(1, 2), (2, 4)], dtype=object)

In [7]: mi.map(lambda x: x)
Out[7]: array([(1, 2), (2, 4)], dtype=object)

In [8]: mi.map(lambda x: x[0])
Out[8]: array([1, 2])

New behavior:

In [61]: idx.map(lambda x: x * 2)
Out[61]: Int64Index([2, 4], dtype='int64')

In [62]: idx.map(lambda x: (x, x * 2))
Out[62]:
map on a Series with datetime64 values may return int64 dtypes rather than int32

| In [65]: s = pd.Series(pd.date_range('2011-01-02T00:00', '2011-01-02T02:00', freq='H'))
| ....:     .tz_localize('Asia/Tokyo'))
| ....: |
| In [66]: s
| Out[66]:
| 0  2011-01-02 00:00:00+09:00
| 1  2011-01-02 01:00:00+09:00
| 2  2011-01-02 02:00:00+09:00
| Length: 3, dtype: datetime64[ns, Asia/Tokyo]

Previous behavior:

| In [9]: s.map(lambda x: x.hour)
| Out[9]:
| 0  0
| 1  1
| 2  2
| dtype: int32

New behavior:

| In [67]: s.map(lambda x: x.hour)
| Out[67]:
| 0  0
| 1  1
| 2  2
| Length: 3, dtype: int64
**Accessing datetime fields of Index now return Index**

The datetime-related attributes (see [here](#) for an overview) of `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex` previously returned numpy arrays. They will now return a new `Index` object, except in the case of a boolean field, where the result will still be a boolean `ndarray`. (GH15022)

Previous behaviour:
```
In [1]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [2]: idx.hour
Out[2]: array([ 0, 10, 20, 6, 16], dtype=int32)
```

New behavior:
```
In [68]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [69]: idx.hour
Out[69]: Int64Index([0, 10, 20, 6, 16], dtype='int64')
```

This has the advantage that specific `Index` methods are still available on the result. On the other hand, this might have backward incompatibilities: e.g. compared to numpy arrays, `Index` objects are not mutable. To get the original `ndarray`, you can always convert explicitly using `np.asarray(idx.hour)`.

**pd.unique will now be consistent with extension types**

In prior versions, using `Series.unique()` and `pandas.unique()` on `Categorical` and tz-aware data-types would yield different return types. These are now made consistent. (GH15903)

- Datetime tz-aware

  Previous behaviour:

  ```
  # Series
  In [5]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
  "...:    pd.Timestamp('20160101', tz='US/Eastern'))].unique()
  Out[5]: array([Timestamp('2016-01-01 00:00:00-05:00', tz='US/Eastern')],
  "...:     _dtype=object)

  In [6]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
  "...:    pd.Timestamp('20160101', tz='US/Eastern'))])
  Out[6]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')

  # Index
  In [7]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
  "...:    pd.Timestamp('20160101', tz='US/Eastern'))].unique()
  Out[7]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/...
  "...:     Eastern'], freq=None)

  In [8]: pd.unique([pd.Timestamp('20160101', tz='US/Eastern'),
  "...:    pd.Timestamp('20160101', tz='US/Eastern'))])
  Out[8]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')
  ```

New behavior:
# Series, returns an array of Timestamp tz-aware

In [70]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                 pd.Timestamp('20160101', tz='US/Eastern')]).unique()

Out[70]:
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

In [71]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                         pd.Timestamp('20160101', tz='US/Eastern')]))

Out[71]:
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

# Index, returns a DatetimeIndex

In [72]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
               pd.Timestamp('20160101', tz='US/Eastern')]).unique()

Out[72]:
DatetimeIndex(['2016-01-01 00:00:00-05:00'],
               dtype=dataframe64[ns, US/Eastern],
               freq=None)

In [73]: pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                         pd.Timestamp('20160101', tz='US/Eastern')]))

Out[73]:
DatetimeIndex(['2016-01-01 00:00:00-05:00'],
               dtype=dataframe64[ns, US/Eastern],
               freq=None)

- Categoricals

 Previous behaviour:

In [1]: pd.Series(list('baabc'), dtype='category').unique()

Out[1]:
[b, a, c]
Categories (3, object): [b, a, c]

In [2]: pd.unique(pd.Series(list('baabc'), dtype='category'))

Out[2]:
array(['b', 'a', 'c'], dtype=object)

New behavior:

# returns a Categorical
In [74]: pd.Series(list('baabc'), dtype='category').unique()

Out[74]:
['b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']

In [75]: pd.unique(pd.Series(list('baabc'), dtype='category'))

Out[75]:
['b', 'a', 'c']
Categories (3, object): ['a', 'b', 'c']
S3 file handling

pandas now uses s3fs for handling S3 connections. This shouldn’t break any code. However, since s3fs is not a required dependency, you will need to install it separately, like boto in prior versions of pandas. (GH11915).

Partial string indexing changes

DatetimeIndex Partial String Indexing now works as an exact match, provided that string resolution coincides with index resolution, including a case when both are seconds (GH14826). See Slice vs. Exact Match for details.

```
In [76]: df = pd.DataFrame({'a': [1, 2, 3]}, pd.DatetimeIndex(['2011-12-31 23:59:59', '2012-01-01 00:00:00', '2012-01-01 00:00:01']))

Previous behavior:

In [4]: df['2011-12-31 23:59:59']
Out[4]:
   a
2011-12-31 23:59:59    1
Name: a, dtype: int64

In [5]: df['a']['2011-12-31 23:59:59']
Out[5]:
2011-12-31 23:59:59    1
Name: a, dtype: int64

New behavior:

In [4]: df['2011-12-31 23:59:59']
KeyError: '2011-12-31 23:59:59'

In [5]: df['a']['2011-12-31 23:59:59']
Out[5]: 1
```

Concat of different float dtypes will not automatically upcast

Previously, concat of multiple objects with different float dtypes would automatically upcast results to a dtype of float64. Now the smallest acceptable dtype will be used (GH13247)

```
In [77]: df1 = pd.DataFrame(np.array([1.0], dtype=np.float32, ndmin=2))

In [78]: df1.dtypes
Out[78]:
0    float32
Length: 1, dtype: object

In [79]: df2 = pd.DataFrame(np.array([np.nan], dtype=np.float32, ndmin=2))

In [80]: df2.dtypes
Out[80]:
```
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

<table>
<thead>
<tr>
<th></th>
<th>float32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Length: 1, dtype: object</td>
<td></td>
</tr>
</tbody>
</table>

Previous behavior:

```
In [7]: pd.concat([df1, df2]).dtypes
Out[7]:
0  float64
dtype: object
```

New behavior:

```
In [81]: pd.concat([df1, df2]).dtypes
Out[81]:
0  float32
Length: 1, dtype: object
```

pandas Google BigQuery support has moved

pandas has split off Google BigQuery support into a separate package pandas-gbq. You can conda install pandas-gbq -c conda-forge or pip install pandas-gbq to get it. The functionality of `read_gbq()` and `DataFrame.to_gbq()` remain the same with the currently released version of pandas-gbq=0.1.4. Documentation is now hosted here (GH15347)

Memory usage for Index is more accurate

In previous versions, showing `.memory_usage()` on a pandas structure that has an index, would only include actual index values and not include structures that facilitated fast indexing. This will generally be different for `Index` and `MultiIndex` and less-so for other index types. (GH15237)

Previous behavior:

```
In [8]: index = pd.Index(['foo', 'bar', 'baz'])
In [9]: index.memory_usage(deep=True)
Out[9]: 180
In [10]: index.get_loc('foo')
Out[10]: 0
In [11]: index.memory_usage(deep=True)
Out[11]: 180
```

New behavior:

```
In [8]: index = pd.Index(['foo', 'bar', 'baz'])
In [9]: index.memory_usage(deep=True)
Out[9]: 180
In [10]: index.get_loc('foo')
```
DataFrame.sort_index changes

In certain cases, calling .sort_index() on a MultiIndexed DataFrame would return the same DataFrame without seeming to sort. This would happen with a lexsorted, but non-monotonic levels. (GH15622, GH15687, GH14015, GH13431, GH15797)

This is unchanged from prior versions, but shown for illustration purposes:

```
In [82]: df = pd.DataFrame(np.arange(6), columns=['value'],
                       index=pd.MultiIndex.from_product([list('BA'), range(3)]))
```

```
value
B   0   0
   1   1
   2   2
A   0   3
   1   4
   2   5
[6 rows x 1 columns]
```

```
In [87]: df.index.is_lexsorted()
Out[87]: False
```

```
In [88]: df.index.is_monotonic
Out[88]: False
```

Sorting works as expected

```
In [84]: df.sort_index()
```

```
value
A   0   3
   1   4
   2   5
B   0   0
   1   1
   2   2
[6 rows x 1 columns]
```

```
In [90]: df.sort_index().index.is_lexsorted()
Out[90]: True
```
In [91]: df.sort_index().index.is_monotonic
Out[91]: True

However, this example, which has a non-monotonic 2nd level, doesn’t behave as desired.

In [85]: df = pd.DataFrame(
             
             ['value': [1, 2, 3, 4]},
             index=pd.MultiIndex(
             [['a', 'b'], ['bb', 'aa']],
             [[0, 0, 1, 1], [0, 1, 0, 1]])
             
In [86]: df
Out[86]:
  value
  a bb 1
  aa 2
  b bb 3
  aa 4

[4 rows x 1 columns]

Previous behavior:

In [11]: df.sort_index()
Out[11]:
  value
  a bb 1
  aa 2
  b bb 3
  aa 4

In [14]: df.sort_index().index.is_lexsorted()
Out[14]: True

In [15]: df.sort_index().index.is_monotonic
Out[15]: False

New behavior:

In [94]: df.sort_index()
Out[94]:
  value
  a aa 2
  bb 1
  b aa 4
  bb 3

[4 rows x 1 columns]

In [95]: df.sort_index().index.is_lexsorted()
Out[95]: True

(continues on next page)
In [96]: df.sort_index().index.is_monotonic
Out[96]: True

GroupBy describe formatting

The output formatting of `groupby.describe()` now labels the `describe()` metrics in the columns instead of the index. This format is consistent with `groupby.agg()` when applying multiple functions at once. (GH4792)

Previous behavior:

In [1]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
In [2]: df.groupby('A').describe()
Out[2]:

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1.500</td>
<td>0.707</td>
<td>1.0</td>
<td>1.25</td>
<td>1.5</td>
<td>1.75</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.500</td>
<td>0.707</td>
<td>3.0</td>
<td>3.25</td>
<td>3.5</td>
<td>3.75</td>
<td>4.0</td>
</tr>
</tbody>
</table>

In [3]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
Out[3]:

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>amin</th>
<th>amax</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.5</td>
<td>0.707</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>0.707</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

New behavior:

In [87]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
In [88]: df.groupby('A').describe()
Out[88]:

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
<td>1.25</td>
<td>1.5</td>
<td>1.75</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.0</td>
<td>2.5</td>
<td>3.0</td>
<td>3.25</td>
<td>3.5</td>
<td>3.75</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Window binary corr/cov operations return a MultiIndex DataFrame

A binary window operation, like .corr() or .cov(), when operating on a .rolling(..), .expanding(..), or .ewm(..) object, will now return a 2-level MultiIndexed DataFrame rather than a Panel, as Panel is now deprecated, see here. These are equivalent in function, but a MultiIndexed DataFrame enjoys more support in pandas. See the section on Windowed Binary Operations for more information. (GH15677)
In [93]: res = df.rolling(12).corr()

In [94]: res.tail()
Out[94]:
   bar  A     B
  foo  bar
2016-04-07   B -0.132090  1.000000
2016-04-08   A  1.000000 -0.145775
          B -0.145775  1.000000
2016-04-09   A  1.000000  0.119645
          B  0.119645  1.000000

[5 rows x 2 columns]

Retrieving a correlation matrix for a cross-section

In [95]: df.rolling(12).corr().loc['2016-04-07']
Out[95]:
   bar  A     B
  bar
  A  1.000000 -0.132090
  B -0.132090  1.000000

[2 rows x 2 columns]

**HDFStore where string comparison**

In previous versions most types could be compared to string column in a HDFStore usually resulting in an invalid comparison, returning an empty result frame. These comparisons will now raise a TypeError (GH15492)

In [96]: df = pd.DataFrame({"unparsed_date": ['2014-01-01', '2014-01-01']})

In [97]: df.to_hdf('store.h5', 'key', format='table', data_columns=True)

In [98]: df.dtypes
Out[98]:
unparsed_date  object
Length: 1, dtype: object

Previous behavior:

In [4]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
File "<string>" line 1
   (unparsed_date > 1970-01-01 00:00:00.388552400)
^ SyntaxError: invalid token

New behavior:

In [18]: ts = pd.Timestamp('2014-01-01')

In [19]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
Index.intersection and inner join now preserve the order of the left Index

`Index.intersection()` now preserves the order of the calling Index (left) instead of the other Index (right) (GH15582). This affects inner joins, `DataFrame.join()` and `merge()`, and the `.align` method.

- **Index.intersection**

  ```python
  In [99]: left = pd.Index([2, 1, 0])
  In [100]: left
  Out[100]: Int64Index([2, 1, 0], dtype='int64')
  In [101]: right = pd.Index([1, 2, 3])
  In [102]: right
  Out[102]: Int64Index([1, 2, 3], dtype='int64')
  Previous behavior:
  In [4]: left.intersection(right)
  Out[4]: Int64Index([1, 2], dtype='int64')
  New behavior:
  In [103]: left.intersection(right)
  Out[103]: Int64Index([2, 1], dtype='int64')
  ```

- **DataFrame.join and pd.merge**

  ```python
  In [104]: left = pd.DataFrame({'a': [20, 10, 0]}, index=[2, 1, 0])
  In [105]: left
  Out[105]:
   a
  2  20
  1  10
  0   0
  [3 rows x 1 columns]
  In [106]: right = pd.DataFrame({'b': [100, 200, 300]}, index=[1, 2, 3])
  In [107]: right
  Out[107]:
   b
  1  100
  2  200
  3  300
  ```
Pivot table always returns a DataFrame

The documentation for `pivot_table()` states that a DataFrame is always returned. Here a bug is fixed that allowed this to return a Series under certain circumstances. (GH4386)

```
In [109]: df = pd.DataFrame({'col1': [3, 4, 5],
                      ....:                 'col2': ['C', 'D', 'E'],
                      ....:                 'col3': [1, 3, 9]})

In [110]: df
Out[110]:
    col1  col2  col3
 0    3    C    1
 1    4    D    3
 2    5    E    9
[3 rows x 3 columns]
```

Old behavior:

```
In [2]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[2]:
     col2  col3
 0   C     3
 1   D     4
 2   E     5
Name: col1, dtype: int64
```

New behavior:

```
In [2]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[2]:
     col2  col3
 3   C     3
 9   E     5
Name: col1, dtype: int64
```
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

```python
In [111]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[111]:
    col1
col3  col2
  1   C     3
  3   D     4
  9   E     5
[3 rows x 1 columns]
```

### Other API changes

- `numexpr` version is now required to be >= 2.4.6 and it will not be used at all if this requisite is not fulfilled (GH15213).
- `CParserError` has been renamed to `ParserError` in `pd.read_csv()` and will be removed in the future (GH12665)
- `SparseArray.cumsum()` and `SparseSeries.cumsum()` will now always return `SparseArray` and `SparseSeries` respectively (GH12855)
- `DataFrame.applymap()` with an empty `DataFrame` will return a copy of the empty `DataFrame` instead of a `Series` (GH8222)
- `Series.map()` now respects default values of dictionary subclasses with a `__missing__` method, such as `collections.Counter` (GH15999)
- `.loc` has compat with `.ix` for accepting iterators, and `NamedTuples` (GH15120)
- `interpolate()` and `fillna()` will raise a `ValueError` if the `limit` keyword argument is not greater than 0. (GH9217)
- `pd.read_csv()` will now issue a `ParserWarning` whenever there are conflicting values provided by the dialect parameter and the user (GH14898)
- `pd.read_csv()` will now raise a `ValueError` for the C engine if the quote character is larger than one byte (GH11592)
- `inplace` arguments now require a boolean value, else a `ValueError` is thrown (GH14189)
- `pandas.api.types.is_datetime64_ns_dtype` will now report `True` on a tz-aware dtype, similar to `pandas.api.types.is_datetime64_any_dtype`
- `DataFrame.asof()` will return a null filled `Series` instead the scalar `NaN` if a match is not found (GH15118)
- Specific support for `copy.copy()` and `copy.deepcopy()` functions on `NDFrame` objects (GH15444)
- `Series.sort_values()` accepts a one element list of bool for consistency with the behavior of `DataFrame.sort_values()` (GH15604)
- `.merge()` and `.join()` on category dtype columns will now preserve the category dtype when possible (GH10409)
- `SparseDataFrame.default_fill_value` will be 0, previously was `nan` in the return from `pd.get_dummies(..., sparse=True)` (GH15594)
- The default behaviour of `Series.str.match` has changed from extracting groups to matching the pattern. The extracting behaviour was deprecated since pandas version 0.13.0 and can be done with the `Series.str.extract` method (GH5224). As a consequence, the `as_indexer` keyword is ignored (no longer needed to specify the new behaviour) and is deprecated.
• NaT will now correctly report False for datetimelike boolean operations such as \texttt{is\_month\_start} (GH15781)
• NaT will now correctly return \texttt{np.nan} for Timedelta and Period accessors such as days and quarter (GH15782)
• NaT will now returns NaT for \texttt{tz\_localize} and \texttt{tz\_convert} methods (GH15830)
• DataFrame and Panel constructors with invalid input will now raise \texttt{ValueError} rather than \texttt{PandasError}, if called with scalar inputs and not axes (GH15541)
• DataFrame and Panel constructors with invalid input will now raise \texttt{ValueError} rather than \texttt{pandas.core.common.PandasError}, if called with scalar inputs and not axes; The exception \texttt{PandasError} is removed as well. (GH15541)
• The exception \texttt{pandas.core.common.AmbiguousIndexError} is removed as it is not referenced (GH15541)

Reorganization of the library: privacy changes

Modules privacy has changed

Some formerly public python/c/c++/cython extension modules have been moved and/or renamed. These are all removed from the public API. Furthermore, the \texttt{pandas.core}, \texttt{pandas_compat}, and \texttt{pandas_util} top-level modules are now considered to be PRIVATE. If indicated, a deprecation warning will be issued if you reference these modules. (GH12588)

<table>
<thead>
<tr>
<th>Previous Location</th>
<th>New Location</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.lib</td>
<td>pandas._libs.lib</td>
<td>X</td>
</tr>
<tr>
<td>pandas.tslib</td>
<td>pandas._libs.tslib</td>
<td>X</td>
</tr>
<tr>
<td>pandas.computation</td>
<td>pandas.core.computation</td>
<td>X</td>
</tr>
<tr>
<td>pandas.msgpack</td>
<td>pandas.io.msgpack</td>
<td></td>
</tr>
<tr>
<td>pandas.index</td>
<td>pandas._libs.index</td>
<td></td>
</tr>
<tr>
<td>pandas.algos</td>
<td>pandas._libs.algos</td>
<td></td>
</tr>
<tr>
<td>pandas hashtable</td>
<td>pandas._libs.hashtable</td>
<td></td>
</tr>
<tr>
<td>pandas indexes</td>
<td>pandas.core.indexes</td>
<td></td>
</tr>
<tr>
<td>pandas.json</td>
<td>pandas._libs.json / pandas.io.json</td>
<td>X</td>
</tr>
<tr>
<td>pandas.parser</td>
<td>pandas._libs.parsers</td>
<td>X</td>
</tr>
<tr>
<td>pandas formats</td>
<td>pandas.io.formats</td>
<td></td>
</tr>
<tr>
<td>pandas.sparse</td>
<td>pandas.core.sparse</td>
<td></td>
</tr>
<tr>
<td>pandas tools</td>
<td>pandas.core.reshape</td>
<td>X</td>
</tr>
<tr>
<td>pandas.types</td>
<td>pandas.core.dtypes</td>
<td>X</td>
</tr>
<tr>
<td>pandas.io.sas.saslib</td>
<td>pandas.io.sas._sas</td>
<td></td>
</tr>
<tr>
<td>pandas join</td>
<td>pandas._libs.join</td>
<td></td>
</tr>
<tr>
<td>pandas_hash</td>
<td>pandas._libs.hashing</td>
<td></td>
</tr>
<tr>
<td>pandas_period</td>
<td>pandas._libs.period</td>
<td></td>
</tr>
<tr>
<td>pandas_sparse</td>
<td>pandas._libs.sparse</td>
<td></td>
</tr>
<tr>
<td>pandas testing</td>
<td>pandas._libs.testing</td>
<td></td>
</tr>
<tr>
<td>pandas_window</td>
<td>pandas._libs.window</td>
<td></td>
</tr>
</tbody>
</table>

Some new subpackages are created with public functionality that is not directly exposed in the top-level namespace: \texttt{pandas.errors}, \texttt{pandas.plotting} and \texttt{pandas.testing} (more details below). Together with \texttt{pandas.api.types} and certain functions in the \texttt{pandas.io} and \texttt{pandas.tseries} submodules, these are now the public subpackages.
Further changes:

- The function `union_categoricals()` is now importable from `pandas.api.types`, formerly from `pandas.types.concat` (GH15998)
- The type import `pandas.tslib.NaTType` is deprecated and can be replaced by using `type(pandas.NaT)` (GH16146)
- The public functions in `pandas.tools.hashing` deprecated from that locations, but are now importable from `pandas.util` (GH16223)
- The modules in `pandas.util`: decorators, print_versions, doctools, validators, depr_module are now private. Only the functions exposed in `pandas.util` itself are public (GH16223)

**pandas.errors**

We are adding a standard public module for all pandas exceptions & warnings `pandas.errors` (GH14800). Previously these exceptions & warnings could be imported from `pandas.core.common` or `pandas.io.common`. These exceptions and warnings will be removed from the *.common locations in a future release. (GH15541)

The following are now part of this API:

```python
['DtypeWarning',
 'EmptyDataError',
 'OutOfBoundsDatetime',
 'ParserError',
 'ParserWarning',
 'PerformanceWarning',
 'UnsortedIndexError',
 'UnsupportedFunctionCall']
```

**pandas.testing**

We are adding a standard module that exposes the public testing functions in `pandas.testing` (GH9895). Those functions can be used when writing tests for functionality using pandas objects.

The following testing functions are now part of this API:

- `testing.assert_frame_equal()`
- `testing.assert_series_equal()`
- `testing.assert_index_equal()`

**pandas.plotting**

A new public `pandas.plotting` module has been added that holds plotting functionality that was previously in either `pandas.tools.plotting` or in the top-level namespace. See the deprecations sections for more details.
Other development changes

- Building pandas for development now requires cython \geq 0.23 (GH14831)
- Require at least 0.23 version of cython to avoid problems with character encodings (GH14699)
- Switched the test framework to use pytest (GH13097)
- Reorganization of tests directory layout (GH14854, GH15707).

Deprecations

Deprecate .ix

The .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers. .ix offers a lot of magic on the inference of what the user wants to do. More specifically, .ix can decide to index positionally OR via labels, depending on the data type of the index. This has caused quite a bit of user confusion over the years. The full indexing documentation is here. (GH14218)

The recommended methods of indexing are:

- .loc if you want to label index
- .iloc if you want to positionally index.

Using .ix will now show a DeprecationWarning with a link to some examples of how to convert code here.

```
In [112]: df = pd.DataFrame({'A': [1, 2, 3],
                     'B': [4, 5, 6]},
                     index=list('abc'))

In [113]: df
Out[113]:
   A  B
a 1  4
b 2  5
c 3  6

[3 rows x 2 columns]
```

Previous behavior, where you wish to get the 0th and the 2nd elements from the index in the ‘A’ column.

```
In [3]: df.ix[[0, 2], 'A']
Out[3]:
a 1
Name: A, dtype: int64
```

Using .loc. Here we will select the appropriate indexes from the index, then use label indexing.

```
In [114]: df.loc[df.index[[0, 2]], 'A']
Out[114]:
a 1
Name: A, Length: 2, dtype: int64
```
Using `.iloc`. Here we will get the location of the ‘A’ column, then use positional indexing to select things.

```
In [115]: df.iloc[[0, 2], df.columns.get_loc('A')]
Out[115]:
a 1
c 3
Name: A, Length: 2, dtype: int64
```

## Deprecate Panel

Panel is deprecated and will be removed in a future version. The recommended way to represent 3-D data are with a MultiIndex on a DataFrame via the `to_frame()` or with the xarray package. pandas provides a `to_xarray()` method to automate this conversion (GH13563).

```
In [133]: import pandas._testing as tm
In [134]: p = tm.makePanel()
In [135]: p
Out[135]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

Convert to a MultiIndex DataFrame

```
```
In [136]: p.to_frame()
Out[136]:
<table>
<thead>
<tr>
<th>major</th>
<th>minor</th>
<th>ItemA</th>
<th>ItemB</th>
<th>ItemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03 A</td>
<td>0.628776</td>
<td>-1.409432</td>
<td>0.209395</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.988138</td>
<td>-1.347533</td>
<td>-0.896581</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-0.938153</td>
<td>1.272395</td>
<td>-0.161137</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>-0.223019</td>
<td>-0.591863</td>
<td>-1.051539</td>
<td></td>
</tr>
<tr>
<td>2000-01-04 A</td>
<td>0.186494</td>
<td>1.422986</td>
<td>-0.592886</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-0.072608</td>
<td>0.363565</td>
<td>1.104352</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-1.239072</td>
<td>-1.449567</td>
<td>0.889157</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2.123692</td>
<td>-0.414505</td>
<td>-0.319561</td>
<td></td>
</tr>
<tr>
<td>2000-01-05 A</td>
<td>0.952478</td>
<td>-2.147855</td>
<td>-1.473116</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-0.550603</td>
<td>-0.014752</td>
<td>-0.431550</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.139683</td>
<td>-1.195524</td>
<td>0.288377</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0.122273</td>
<td>-1.425795</td>
<td>-0.619993</td>
<td></td>
</tr>
</tbody>
</table>

[12 rows x 3 columns]
```

Convert to an xarray DataArray

```
In [137]: p.to_xarray()
Out[137]:
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[0.628776, 0.988138, -0.938153, -0.223019],
       [0.186494, -0.072608, -1.239072, 2.123692],
       [0.952478, -0.550603, 0.139683, 0.122273]])
```
Deprecate groupby.agg() with a dictionary when renaming

The .groupby(..).agg(..), .rolling(..).agg(..), and .resample(..).agg(..) syntax can accept a variable of inputs, including scalars, list, and a dict of column names to scalars or lists. This provides a useful syntax for constructing multiple (potentially different) aggregations.

However, .agg(..) can also accept a dict that allows ‘renaming’ of the result columns. This is a complicated and confusing syntax, as well as not consistent between Series and DataFrame. We are deprecating this ‘renaming’ functionality.

- We are deprecating passing a dict to a grouped/rolled/resampled Series. This allowed one to rename the resulting aggregation, but this had a completely different meaning than passing a dictionary to a grouped DataFrame, which accepts column-to-aggregations.

- We are deprecating passing a dict-of-dicts to a grouped/rolled/resampled DataFrame in a similar manner.

This is an illustrative example:

```python
In [116]: df = pd.DataFrame({'A': [1, 1, 1, 2, 2],
    ....:     'B': range(5),
    ....:     'C': range(5)})

In [117]: df
Out[117]:
     A  B  C
0    0  0  0
1    1  1  1
2    2  1  2
3    3  2  3
4    4  2  4
[5 rows x 3 columns]
```

Here is a typical useful syntax for computing different aggregations for different columns. This is a natural, and useful syntax. We aggregate from the dict-to-list by taking the specified columns and applying the list of functions. This returns a MultiIndex for the columns (this is not deprecated).
In [118]: df.groupby('A').agg({'B': 'sum', 'C': 'min'})

Out[118]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

[2 rows x 2 columns]

Here's an example of the first deprecation, passing a dict to a grouped Series. This is a combination aggregation & renaming:

In [6]: df.groupby('A').B.agg({'foo': 'count'})

FutureWarning: using a dict on a Series for aggregation is deprecated and will be removed in a future version

Out[6]:

<table>
<thead>
<tr>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

You can accomplish the same operation, more idiomatically by:

In [119]: df.groupby('A').B.agg({'foo': 'sum'}).rename(columns={'count': 'foo'})

Out[119]:

<table>
<thead>
<tr>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

[2 rows x 1 columns]

Here's an example of the second deprecation, passing a dict-of-dict to a grouped DataFrame:

In [23]: (df.groupby('A')

...:   .agg({'B': {'foo': 'sum'}, 'C': {'bar': 'min'}})

...: )

FutureWarning: using a dict with renaming is deprecated and will be removed in a future version

Out[23]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

[2 rows x 1 columns]

You can accomplish nearly the same by:

In [120]: (df.groupby('A')

......:   .agg({'B': 'sum', 'C': 'min'})

......:   .rename(columns={'B': 'foo', 'C': 'bar'})

......: )

(continues on next page)
Deprecate `.plotting`

The `pandas.tools.plotting` module has been deprecated, in favor of the top level `pandas.plotting` module. All the public plotting functions are now available from `pandas.plotting` (GH12548).

Furthermore, the top-level `pandas.scatter_matrix` and `pandas.plot_params` are deprecated. Users can import these from `pandas.plotting` as well.

Previous script:

```python
pd.tools.plotting.scatter_matrix(df)
pd.scatter_matrix(df)
```

Should be changed to:

```python
pd.plotting.scatter_matrix(df)
```

Other deprecations

- `SparseArray.to_dense()` has deprecated the `fill` parameter, as that parameter was not being respected (GH14647)
- `SparseSeries.to_dense()` has deprecated the `sparse_only` parameter (GH14647)
- `Series.repeat()` has deprecated the `reps` parameter in favor of `repeats` (GH12662)
- The `Series` constructor and `.astype` method have deprecated accepting timestamp dtypes without a frequency (e.g. `np.datetime64`) for the `dtype` parameter (GH15524)
- `Index.repeat()` and `MultiIndex.repeat()` have deprecated the `n` parameter in favor of `repeats` (GH12662)
- `Categorical.searchsorted()` and `Series.searchsorted()` have deprecated the `v` parameter in favor of `value` (GH12662)
- `TimedeltaIndex.searchsorted()`, `DatetimeIndex.searchsorted()`, and `PeriodIndex.searchsorted()` have deprecated the `key` parameter in favor of `value` (GH12662)
- `DataFrame.astype()` has deprecated the `raise_on_error` parameter in favor of `errors` (GH14878)
- `Series.sortlevel` and `DataFrame.sortlevel` have been deprecated in favor of `Series.sort_index` and `DataFrame.sort_index` (GH15099)
- Importing `concat` from `pandas.tools.merge` has been deprecated in favor of imports from the `pandas` namespace. This should only affect explicit imports (GH15358)
• Series/DataFrame/Panel.consolidate() been deprecated as a public method. (GH15483)
• The as_indexer keyword of Series.str.match() has been deprecated (ignored keyword) (GH15257).
• The following top-level pandas functions have been deprecated and will be removed in a future version (GH13790, GH15940)
  – pd.pnow(), replaced by Period.now()
  – pd.Term, is removed, as it is not applicable to user code. Instead use in-line string expressions in the where clause when searching in HDFStore
  – pd.Expr, is removed, as it is not applicable to user code.
  – pd.match(), is removed.
  – pd.groupby(), replaced by using the .groupby() method directly on a Series/DataFrame
  – pd.get_store(), replaced by a direct call to pd.HDFStore(...)
• is_any_int_dtype, is_floating_dtype, and is_sequence are deprecated from pandas.api.types (GH16042)

Removal of prior version deprecations/changes

• The pandas.rpy module is removed. Similar functionality can be accessed through the rpy2 project. See the R interfacing docs for more details.
• The pandas.io.ga module with a google-analytics interface is removed (GH11308). Similar functionality can be found in the Google2Pandas package.
• pd.to_datetime and pd.to_timedelta have dropped the coerce parameter in favor of errors (GH13602)
• pandas.stats.fama_macbeth, pandas.stats.ols, pandas.stats.plm and pandas.stats.var, as well as the top-level pandas.fama_macbeth and pandas.ols routines are removed. Similar functionality can be found in the statsmodels package. (GH11898)
• The TimeSeries and SparseTimeSeries classes, aliases of Series and SparseSeries, are removed (GH10890, GH15098).
• Series.is_time_series is dropped in favor of Series.index.is_all_dates (GH15098)
• The deprecated irow, icol, iget and iget_value methods are removed in favor of iloc and iat as explained here (GH10711).
• The deprecated DataFrame.iterkv() has been removed in favor of DataFrame.iteritems() (GH10711)
• The Categorical constructor has dropped the name parameter (GH10632)
• Categorical has dropped support for NaN categories (GH10748)
• The take_last parameter has been dropped from duplicated(), drop_duplicates(), nlargest(), and nsmallest() methods (GH10236, GH10792, GH10920)
• Series, Index, and DataFrame have dropped the sort and order methods (GH10726)
• Where clauses in pytables are only accepted as strings and expressions types and not other data-types (GH12027)
• DataFrame has dropped the combineAdd and combineMult methods in favor of add and mul respectively (GH10735)
Performance improvements

- Improved performance of `pd.wide_to_long()` (GH14779)
- Improved performance of `pd.factorize()` by releasing the GIL with object dtype when inferred as strings (GH14859, GH16057)
- Improved performance of timeseries plotting with an irregular DatetimeIndex (or with `compat_x=True`) (GH15073).
- Improved performance of `groupby().cummin()` and `groupby().cummax()` (GH15048, GH15109, GH15561, GH15635)
- Improved performance and reduced memory when indexing with a `MultiIndex` (GH15245)
- When reading buffer object in `read_sas()` method without specified format, filepath string is inferred rather than buffer object. (GH14947)
- Improved performance of `.rank()` for categorical data (GH15498)
- Improved performance when using `.unstack()` (GH15503)
- Improved performance of merge/join on category columns (GH10409)
- Improved performance of `drop_duplicates()` on bool columns (GH12963)
- Improve performance of `pd.core.groupby.GroupBy.apply` when the applied function used the `.name` attribute of the group DataFrame (GH15062).
- Improved performance of `iloc` indexing with a list or array (GH15504).
- Improved performance of `Series.sort_index()` with a monotonic index (GH15694)
- Improved performance in `pd.read_csv()` on some platforms with buffered reads (GH16039)

Bug fixes

Conversion

- Bug in `Timestamp.replace` now raises `TypeError` when incorrect argument names are given; previously this raised `ValueError` (GH15240)
- Bug in `Timestamp.replace` with compat for passing long integers (GH15030)
- Bug in `Timestamp` returning UTC based time/date attributes when a timezone was provided (GH13303, GH6538)
- Bug in `Timestamp` incorrectly localizing timezones during construction (GH11481, GH15777)
- Bug in `TimedeltaIndex` addition where overflow was being allowed without error (GH14816)
- Bug in `TimedeltaIndex` raising a `ValueError` when boolean indexing with `loc` (GH14946)
- Bug in catching an overflow in `Timestamp + Timedelta/Offset` operations (GH15126)
- Bug in `DatetimeIndex.round()` and `Timestamp.round()` floating point accuracy when rounding by milliseconds or less (GH14440, GH15578)
- Bug in `astype()` where `inf` values were incorrectly converted to integers. Now raises error now with `astype()` for Series and DataFrames (GH14265)
- Bug in `DataFrame(..).apply(to_numeric)` when values are of type decimal.Decimal. (GH14827)
- Bug in `describe()` when passing a numpy array which does not contain the median to the `percentiles` keyword argument (GH14908)
- Cleaned up `PeriodIndex` constructor, including raising on floats more consistently (GH13277)
- Bug in using `__deepcopy__` on empty NDFrame objects (GH15370)
- Bug in `.replace()` may result in incorrect dtypes. (GH12747, GH15765)
- Bug in `Series.replace` and `DataFrame.replace` which failed on empty replacement dicts (GH15289)
- Bug in `Series.replace` which replaced a numeric by string (GH15743)
- Bug in `Index` construction with NaN elements and integer dtype specified (GH15187)
- Bug in `Series` construction with a datetimetz (GH14928)
- Bug in `Series.dt.round()` inconsistent behaviour on NaT 's with different arguments (GH14940)
- Bug in `Series` constructor when both `copy=True` and `dtype` arguments are provided (GH15125)
- Incorrectly typed `Series` was returned by comparison methods (e.g., `lt`, `gt`, ...) against a constant for an empty `DataFrame` (GH15077)
- Bug in `Series.ffill()` with mixed dtypes containing tz-aware datetimes. (GH14956)
- Bug in `DataFrame.fillna()` where the argument `downcast` was ignored when `fillna` value was of type `dict` (GH15277)
- Bug in `.asfreq()`, where frequency was not set for empty `Series` (GH14320)
- Bug in `DataFrame` construction with nulls and datetimes in a list-like (GH15869)
- Bug in `DataFrame.fillna()` with tz-aware datetimes (GH15855)
- Bug in `is_string_dtype`, `is_timedelta64_ns_dtype`, and `is_string_like_dtype` in which an error was raised when `None` was passed in (GH15941)
- Bug in the return type of `pd.unique` on a `Categorical`, which was returning an ndarray and not a `Categorical` (GH15093)
- Bug in `Index.to_series()` where the index was not copied (and so mutating later would change the original), (GH15994)
- Bug in indexing with partial string indexing with a len-1 `DataFrame` (GH16071)
- Bug in `Series` construction where passing invalid dtype didn’t raise an error. (GH15520)

**Indexing**

- Bug in `Index` power operations with reversed operands (GH14973)
- Bug in `DataFrame.sort_values()` when sorting by multiple columns where one column is of type `int64` and contains NaT (GH14922)
- Bug in `DataFrame.reindex()` in which `method` was ignored when passing `columns` (GH14992)
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a `Series` indexer (GH14730, GH15424)
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a numpy array (GH15434)
- Bug in `Series.asof` which raised if the series contained all `np.nan` (GH15713)
- Bug in `.at` when selecting from a tz-aware column (GH15822)
• Bug in `Series.where()` and `DataFrame.where()` where array-like conditionals were being rejected (GH15414)
• Bug in `Series.where()` where TZ-aware data was converted to float representation (GH15701)
• Bug in `.loc` that would not return the correct dtype for scalar access for a DataFrame (GH11617)
• Bug in output formatting of a `MultiIndex` when names are integers (GH12223, GH15262)
• Bug in `Categorical.searchsorted()` where alphabetical instead of the provided categorical order was used (GH14522)
• Bug in `Series.iloc` where a `Categorical` object for list-like indexes input was returned, where a `Series` was expected. (GH14580)
• Bug in `DataFrame.isin` comparing datetimelike to empty frame (GH15473)
• Bug in `.reset_index()` when an all NaN level of a `MultiIndex` would fail (GH6322)
• Bug in `.reset_index()` when raising error for index name already present in `MultiIndex` columns (GH16120)
• Bug in creating a `MultiIndex` with tuples and not passing a list of names; this will now raise `ValueError` (GH15110)
• Bug in the HTML display with a `MultiIndex` and truncation (GH14882)
• Bug in the display of `.info()` where a qualifier (+) would always be displayed with a `MultiIndex` that contains only non-strings (GH15245)
• Bug in `pd.concat()` where the names of `MultiIndex` of resulting `DataFrame` are not handled correctly when `None` is presented in the names of `MultiIndex` of input `DataFrame` (GH15787)
• Bug in `DataFrame.sort_index()` and `Series.sort_index()` where `na_position` doesn't work with a `MultiIndex` (GH14784, GH16604)
• Bug in `pd.concat()` when combining objects with a `CategoricalIndex` (GH16111)
• Bug in indexing with a scalar and a `CategoricalIndex` (GH16123)

**IO**

• Bug in `pd.to_numeric()` in which float and unsigned integer elements were being improperly casted (GH14941, GH15005)
• Bug in `pd.read_fwf()` where the skiprows parameter was not being respected during column width inference (GH11256)
• Bug in `pd.read_csv()` in which the dialect parameter was not being verified before processing (GH14898)
• Bug in `pd.read_csv()` in which missing data was being improperly handled with usecols (GH6710)
• Bug in `pd.read_csv()` in which a file containing a row with many columns followed by rows with fewer columns would cause a crash (GH14125)
• Bug in `pd.read_csv()` for the C engine where usecols were being indexed incorrectly with parse_dates (GH14792)
• Bug in `pd.read_csv()` with parse_dates when multi-line headers are specified (GH15376)
• Bug in `pd.read_csv()` with `float_precision= \text{'round_trip'}` which caused a segfault when a text entry is parsed (GH15140)
• Bug in `pd.read_csv()` when an index was specified and no values were specified as null values (GH15835)
• Bug in `pd.read_csv()` in which certain invalid file objects caused the Python interpreter to crash (GH15337)
• Bug in `pd.read_csv()` in which invalid values for `nrows` and `chunksize` were allowed (GH15767)
• Bug in `pd.read_csv()` for the Python engine in which unhelpful error messages were being raised when parsing errors occurred (GH15910)
• Bug in `pd.read_csv()` in which the `skipfooter` parameter was not being properly validated (GH15925)
• Bug in `pd.to_csv()` in which there was numeric overflow when a timestamp index was being written (GH15982)
• Bug in `pd.util.hashing.hash_pandas_object()` in which hashing of categoricals depended on the ordering of categories, instead of just their values. (GH15143)
• Bug in `.to_json()` where `lines=True` and contents (keys or values) contain escaped characters (GH15096)
• Bug in `.to_json()` causing single byte ascii characters to be expanded to four byte unicode (GH15344)
• Bug in `.to_json()` for the C engine where rollover was not correctly handled for case where frac is odd and diff is exactly 0.5 (GH15716, GH15864)
• Bug in `pd.read_json()` for Python 2 where `lines=True` and contents contain non-ascii unicode characters (GH15132)
• Bug in `pd.read_msgpack()` in which `Series` categoricals were being improperly processed (GH14901)
• Bug in `pd.read_msgpack()` which did not allow loading of a dataframe with an index of type `CategoricalIndex` (GH15487)
• Bug in `pd.read_msgpack()` when deserializing a `CategoricalIndex` (GH15487)
• Bug in `DataFrame.to_records()` with converting a `DatetimeIndex` with a timezone (GH13937)
• Bug in `DataFrame.to_records()` which failed with unicode characters in column names (GH11879)
• Bug in `.to_sql()` when writing a `DataFrame` with numeric index names (GH15404).
• Bug in `DataFrame.to_html()` with index=False and max_rows raising in `IndexError` (GH14998)
• Bug in `pd.read_hdf()` passing a `Timestamp` to the where parameter with a non date column (GH15492)
• Bug in `DataFrame.to_stata()` and `StataWriter` which produces incorrectly formatted files to be produced for some locales (GH13856)
• Bug in `StataReader` and `StataWriter` which allows invalid encodings (GH15723)
• Bug in the `Series` repr not showing the length when the output was truncated (GH15962).

Plotting

• Bug in `DataFrame.hist` where `plt.tight_layout` caused an `AttributeError` (use `matplotlib` >= 2.0.1) (GH9351)
• Bug in `DataFrame.boxplot` where `fontsize` was not applied to the tick labels on both axes (GH15108)
• Bug in the date and time converters pandas registers with matplotlib not handling multiple dimensions (GH16026)
• Bug in `pd.scatter_matrix()` could accept either `color` or `c`, but not both (GH14855)
pandas: powerful Python data analysis toolkit, Release 1.4.2

GroupBy/resample/rolling

- Bug in `.groupby(..).resample()` when passed the on= kwarg. (GH15021)
- Properly set __name__ and __qualname__ for Groupby.* functions (GH14620)
- Bug in GroupBy.get_group() failing with a categorical grouper (GH15155)
- Bug in `.groupby(...).rolling(...)` when on is specified and using a DatetimeIndex (GH15130, GH13966)
- Bug in groupby operations with timedelta64 when passing numeric_only=False (GH5724)
- Bug in `groupby.apply()` coercing object dtypes to numeric types, when not all values were numeric (GH14423, GH15421, GH15670)
- Bug in `resample`, where a non-string offset argument would not be applied when resampling a timeseries (GH13218)
- Bug in `DataFrame.groupby().describe()` when grouping on Index containing tuples (GH14848)
- Bug in `groupby().nunique()` with a datetimelike-grouper where bins counts were incorrect (GH13453)
- Bug in `groupby.transform()` that would coerce the resultant dtypes back to the original (GH10972, GH11444)
- Bug in `groupby.agg()` incorrectly localizing timezone on datetime (GH15426, GH10668, GH13046)
- Bug in `.rolling/expanding()` functions where count() was not counting np.Inf, nor handling object dtypes (GH12541)
- Bug in `.rolling()` where pd.Timedelta or datetime.timedelta was not accepted as a window argument (GH15440)
- Bug in `Rolling.quantile` function that caused a segmentation fault when called with a quantile value outside of the range [0, 1] (GH15463)
- Bug in `DataFrame.resample().median()` if duplicate column names are present (GH14233)

Sparse

- Bug in SparseSeries.reindex on single level with list of length 1 (GH15447)
- Bug in repr-formatting a SparseDataFrame after a value was set on (a copy of) one of its series (GH15488)
- Bug in SparseDataFrame construction with lists not coercing to dtype (GH15682)
- Bug in sparse array indexing in which indices were not being validated (GH15863)

Reshaping

- Bug in pd.merge_asof() where left_index or right_index caused a failure when multiple by was specified (GH15676)
- Bug in pd.merge_asof() where left_index/right_index together caused a failure when tolerance was specified (GH15135)
- Bug in `DataFrame.pivot_table()` where dropna=True would not drop all-NaN columns when the columns was a category dtypes (GH15193)
- Bug in pd.melt() where passing a tuple value for value_vars caused a TypeError (GH15348)
- Bug in pd.pivot_table() where no error was raised when values argument was not in the columns (GH14938)
pandas: powerful Python data analysis toolkit, Release 1.4.2

• Bug in pd.concat() in which concatenating with an empty dataframe with join='inner' was being improperly handled (GH15328)
• Bug with sort=True in DataFrame.join and pd.merge when joining on indexes (GH15582)
• Bug in DataFrame.nsmallest and DataFrame.nlargest where identical values resulted in duplicated rows (GH15297)
• Bug in pandas.pivot_table() incorrectly raising UnicodeError when passing unicode input for margins keyword (GH13292)

Numeric

• Bug in .rank() which incorrectly ranks ordered categories (GH15420)
• Bug in .corr() and .cov() where the column and index were the same object (GH14617)
• Bug in .mode() where mode was not returned if was only a single value (GH15714)
• Bug in pd.cut() with a single bin on an all 0s array (GH15428)
• Bug in pd.qcut() with a single quantile and an array with identical values (GH15431)
• Bug in pandas.tools.utils.cartesian_product() with large input can cause overflow on windows (GH15265)
• Bug in .eval() which caused multi-line evals to fail with local variables not on the first line (GH15342)

Other

• Compat with SciPy 0.19.0 for testing on .interpolate() (GH15662)
• Compat for 32-bit platforms for .qcut/cut; bins will now be int64 dtype (GH14866)
• Bug in interactions with Qt when a QtApplication already exists (GH14372)
• Avoid use of np.finfo() during import pandas removed to mitigate deadlock on Python GIL misuse (GH14641)

Contributors

A total of 204 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Adam J. Stewart +
• Adrian +
• Ajay Saxena
• Akash Tandon +
• Albert Villanova del Moral +
• Aleksey Bilogur +
• Alexis Mignon +
• Amol Kahat +
• Andreas Winkler +
• Andrew Kittredge +
• Anthonios Partheniou
• Arco Bast +
• Ashish Singal +
• Baurzhan Muftakhidinov +
• Ben Kandel
• Ben Thayer +
• Ben Welsh +
• Bill Chambers +
• Brandon M. Burroughs
• Brian +
• Brian McFee +
• Carlos Souza +
• Chris
• Chris Ham
• Chris Warth
• Christoph Gohlke
• Christoph Paulik +
• Christopher C. Aycock
• Clemens Brunner +
• D.S. McNeil +
• DaanVanHauwermeiren +
• Daniel Himmelstein
• Dave Willmer
• David Cook +
• David Gwynne +
• David Hoffman +
• David Krych
• Diego Fernandez +
• Dimitris Spathis +
• Dmitry L +
• Dody Suria Wijaya +
• Dominik Stanczak +
• Dr-Irv
• Dr. Irv +
• Elliott Sales de Andrade +
• Ennemoser Christoph +
• Francesc Alted +
• Fumito Hamamura +
• Giacomo Ferroni
• Graham R. Jeffries +
• Greg Williams +
• Guilherme Beltramin +
• Guilherme Samora +
• Hao Wu +
• Harshit Patni +
• Ilya V. Schurov +
• Iván Vallés Pérez
• Jackie Leng +
• Jaehoon Hwang +
• James Draper +
• James Goppert +
• James McBride +
• James Santucci +
• Jan Schulz
• Jeff Carey
• Jeff Reback
• JennaVergeynst +
• Jim +
• Jim Crist
• Joe Jevnik
• Joel Nothman +
• John +
• John Tucker +
• John W. O’Brien
• John Zwinick
• Jon M. Mease
• Jon Mease
• Jonathan Whitmore +
• Jonathan de Bruin +
• Joost Kranendonk +
• Joris Van den Bossche
• Joshua Bradt +
• Julian Santander
• Julien Marrec +
• Jun Kim +
• Justin Solinsky +
• Kacawi +
• Kamal Kamalaldin +
• Kerby Shedden
• Kernc
• Keshav Ramaswamy
• Kevin Sheppard
• Kyle Kelley
• Larry Ren
• Leon Yin +
• Line Pedersen +
• Lorenzo Cestaro +
• Luca Scarabello
• Lukasz +
• Mahmoud Lababidi
• Mark Mandel +
• Matt Roeschke
• Matthew Brett
• Matthew Roeschke +
• Matti Picus
• Maximilian Roos
• Michael Charlton +
• Michael Felt
• Michael Lamparski +
• Michiel Stock +
• Mikolaj Chwalisz +
• Min RK
• Miroslav Šedivý +
• Mykola Golubyev
• Nate Yoder
• Nathalie Rud +
• Nicholas Ver Halen
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Nick Chmura +
- Nolan Nichols +
- Pankaj Pandey +
- Pawel Kordek
- Pete Huang +
- Peter +
- Peter Csizsek +
- Petio Petrov +
- Phil Ruffwind +
- Pietro Battiston
- Piotr Chromiec
- Prasanth Prakash +
- Rob Forgione +
- Robert Bradshaw
- Robin +
- Rodolfo Fernandez
- Roger Thomas
- Rouz Azari +
- Sahil Dua
- Sam Foo +
- Sami Salonen +
- Sarah Bird +
- Sarma Tangirala +
- Scott Sanderson
- Sebastian Bank
- Sebastian Gsänger +
- Shawn Heide
- Shyam Saladi +
- Sinhrks
- Stephen Rauch +
- Sébastien de Menten +
- Tara Adiseshan
- Thiago Serafim
- Thoralf Gutierrez +
- Thrasibule +
- Tobias Gustafsson +
• Tom Augspurger
• Tong SHEN +
• Tong Shen +
• TrigonaMinima +
• Uwe +
• Wes Turner
• Wiktor Tomczak +
• WillAyd
• Yaroslav Halchenko
• Yimeng Zhang +
• abaldenko +
• adrian-stepien +
• alexandercbbooth +
• atbd +
• bastewart +
• bmagnusson +
• carlosdanielcsantos +
• chaimdemulder +
• chris-b1
• dickreuter +
• discort +
• dr-leo +
• dubourg
• dwkenefick +
• funnycrab +
• gfyounq
• goldenbull +
• hesham.shabana@hotmail.com
• jojomdt +
• linebp +
• manu +
• manuels +
• mattip +
• maxalbert +
• mcocdawc +
• nuffe +
pandas: powerful Python data analysis toolkit, Release 1.4.2

- paul-mannino
- pbreach +
- sakkemo +
- scls19fr
- sinhhrs
- stijnvanhoey +
- the-nose-knows +
- themrmax +
- tomrod +
- tzinckgraf
- wandersoncferreira
- watercrossing +
- wcwagner
- xgdgsc +
- yui-knk

5.12 Version 0.19

5.12.1 Version 0.19.2 (December 24, 2016)

This is a minor bug-fix release in the 0.19.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:
- Compatibility with Python 3.6
- Added a Pandas Cheat Sheet. (GH13202).

What's new in v0.19.2

- Enhancements
- Performance improvements
- Bug fixes
- Contributors
Enhancements

The `pd.merge_asof()`, added in 0.19.0, gained some improvements:

- `pd.merge_asof()` gained `left_index/right_index` and `left_by/right_by` arguments (GH14253)
- `pd.merge_asof()` can take multiple columns in by parameter and has specialized dtypes for better performance (GH13936)

Performance improvements

- Performance regression with `PeriodIndex` (GH14822)
- Performance regression in indexing with `getitem` (GH14930)
- Improved performance of `.replace()` (GH12745)
- Improved performance `Series` creation with a `datetime` index and dictionary data (GH14894)

Bug fixes

- Compat with python 3.6 for pickling of some offsets (GH14685)
- Compat with python 3.6 for some indexing exception types (GH14684, GH14689)
- Compat with python 3.6 for deprecation warnings in the test suite (GH14681)
- Compat with python 3.6 for `Timestamp` pickles (GH14689)
- Compat with `dateutil==2.6.0`; segfault reported in the testing suite (GH14621)
- Allow `nanoseconds` in `Timestamp.replace` as a kwarg (GH14621)
- Bug in `pd.read_csv` in which aliasing was being done for `na_values` when passed in as a dictionary (GH14203)
- Bug in `pd.read_csv` in which column indices for a dict-like `na_values` were not being respected (GH14203)
- Bug in `pd.read_csv` where reading files fails, if the number of headers is equal to the number of lines in the file (GH14515)
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when multi-char delimiters were not being respected with quotes (GH14582)
- Fix bugs (GH14734, GH13654) in `pd.read_sas` and `pandas.io.sas.sas7bdat.SAS7BDATReader` that caused problems when reading a SAS file incrementally.
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when `skipfooter` was not being respected by Python’s CSV library (GH13879)
- Bug in `.fillna()` in which timezone aware datetimes were incorrectly rounded (GH14872)
- Bug in `.groupby(..., sort=True)` of a non-lexsorted `MultiIndex` when grouping with multiple levels (GH14776)
- Bug in `pd.cut` with negative values and a single bin (GH14652)
- Bug in `pd.to_numeric` where a 0 was not unsigned on a `downcast='unsigned'` argument (GH14401)
- Bug in plotting regular and irregular timeseries using shared axes (`sharex=True` or `ax.twinx()`) (GH13341, GH14322).
- Bug in not propagating exceptions in parsing invalid datetimes, noted in python 3.6 (GH14561)
**pandas: powerful Python data analysis toolkit, Release 1.4.2**

- Bug in resampling a `DatetimeIndex` in local TZ, covering a DST change, which would raise `AmbiguousTimeError` (GH14682)
- Bug in indexing that transformed `RecursionError` into `KeyError` or `IndexingError` (GH14554)
- Bug in `HDFStore` when writing a `MultiIndex` when using `data_columns=True` (GH14435)
- Bug in `HDFStore.append()` when writing a `Series` and passing a `min_itemsize` argument containing a value for the index (GH11412)
- Bug when writing to a `HDFStore` in table format with a `min_itemsize` value for the index and without asking to append (GH10381)
- Bug in `Series.groupby.nunique()` raising an `IndexError` for an empty `Series` (GH12553)
- Bug in `DataFrame.nlargest` and `DataFrame.nsmallest` when the index had duplicate values (GH13412)
- Bug in clipboard functions on Linux with Python2 with unicode and separators (GH13747)
- Bug in clipboard functions on Windows 10 and Python 3 (GH14362, GH12807)
- Bug in `.to_clipboard()` and Excel compat (GH12529)
- Bug in `DataFrame.combine_first()` for integer columns (GH14687).
- Bug in `pd.read_csv()` in which the `dtype` parameter was not being respected for empty data (GH14712)
- Bug in `pd.read_csv()` in which the `nrows` parameter was not being respected for large input when using the C engine for parsing (GH17626)
- Bug in `pd.merge_asof()` could not handle timezone-aware `DatetimeIndex` when a tolerance was specified (GH14844)
- Explicit check in `to_stata` and `StataWriter` for out-of-range values when writing doubles (GH14618)
- Bug in `.plot(kind='kde')` which did not drop missing values to generate the KDE Plot, instead generating an empty plot. (GH14821)
- Bug in `unstack()` if called with a list of column(s) as an argument, regardless of the dtypes of all columns, they get coerced to `object` (GH11847)

**Contributors**

A total of 33 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Ajay Saxena +
- Ben Kandel
- Chris
- Chris Ham +
- Christopher C. Aycock
- Daniel Himmelstein +
- Dave Willmer +
- Dr-Irv
- Jeff Carey +
- Jeff Reback
- Joe Jevnik
5.12.2 Version 0.19.1 (November 3, 2016)

This is a minor bug-fix release from 0.19.0 and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

What’s new in v0.19.1

- Performance improvements
- Bug fixes
- Contributors
Performance improvements

- Fixed performance regression in factorization of `Period` data (GH14338)
- Fixed performance regression in `Series.asof(where)` when `where` is a scalar (GH14461)
- Improved performance in `DataFrame.asof(where)` when `where` is a scalar (GH14461)
- Improved performance in `.to_json()` when `lines=True` (GH14408)
- Improved performance in certain types of `loc` indexing with a MultiIndex (GH14551).

Bug fixes

- Source installs from PyPI will now again work without cython installed, as in previous versions (GH14204)
- Compat with Cython 0.25 for building (GH14496)
- Fixed regression where user-provided file handles were closed in `read_csv (c engine)` (GH14418).
- Fixed regression in `DataFrame.quantile` when missing values where present in some columns (GH14357).
- Fixed regression in `Index.difference` where the `freq` of a `DatetimeIndex` was incorrectly set (GH14323)
- Added back `pandas.core.common.array_equivalent` with a deprecation warning (GH14555).
- Bug in `pd.read_csv` for the C engine in which quotation marks were improperly parsed in skipped rows (GH14459)
- Bug in `pd.read_csv` for Python 2.x in which Unicode quote characters were no longer being respected (GH14477)
- Fixed regression in `Index.append` when categorical indices were appended (GH14545).
- Fixed regression in `pd.DataFrame` where constructor fails when given dict with `None` value (GH14381)
- Fixed regression in `DatetimeIndex._maybe_cast_slice_bound` when index is empty (GH14354).
- Bug in localizing an ambiguous timezone when a boolean is passed (GH14402)
- Bug in `TimedeltaIndex` addition with a Datetime-like object where addition overflow in the negative direction was not being caught (GH14068, GH14453)
- Bug in string indexing against data with `object` Index may raise `AttributeError` (GH14424)
- Correctly raise `ValueError` on empty input to `pd.eval()` and `df.query()` (GH13139)
- Bug in `RangeIndex.intersection` when result is a empty set (GH14364).
- Bug in groupby-transform broadcasting that could cause incorrect dtype coercion (GH14457)
- Bug in `Series.__setitem__` which allowed mutating read-only arrays (GH14359).
- Bug in `DataFrame.insert` where multiple calls with duplicate columns can fail (GH14291)
- `pd.merge()` will raise `ValueError` with non-boolean parameters in passed boolean type arguments (GH14434)
- Bug in `Timestamp` where dates very near the minimum (1677-09) could underflow on creation (GH14415)
- Bug in `pd.concat` where names of the keys were not propagated to the resulting `MultiIndex` (GH14252)
- Bug in `pd.concat` where `axis` cannot take string parameters 'rows' or 'columns' (GH14369)
- Bug in `pd.concat` with dataframes heterogeneous in length and tuple keys (GH14438)
- Bug in `MultiIndex.set_levels` where illegal level values were still set after raising an error (GH13754)
- Bug in `DataFrame.to_json` where `lines=True` and a value contained a `}` character (GH14391)
• Bug in `df.groupby` causing an `AttributeError` when grouping a single index frame by a column and the index level (GH14327)
• Bug in `df.groupby` where `TypeError` raised when `pd.Grouper(key=...)` is passed in a list (GH14334)
• Bug in `pd.pivot_table` may raise `TypeError` or `ValueError` when `index` or `columns` is not scalar and `values` is not specified (GH14380)

Contributors

A total of 30 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Adam Chainz +
• Anthonios Partheniou
• Arash Rouhani +
• Ben Kandel
• Brandon M. Burroughs +
• Chris
• Chris Warth
• David Krych +
• Iván Vallés Pérez +
• Jeff Reback
• Joe Jevnik
• Jon M. Mease +
• Jon Mease +
• Joris Van den Bossche
• Josh Owen +
• Keshav Ramaswamy +
• Larry Ren +
• Michael Felt +
• Piotr Chromiec +
• Robert Bradshaw +
• Sinhrks
• Thiago Serafim +
• Tom Bird
• bkandel +
• chris-b1
• dubourg +
• gfyoungh
• mattrijk +
5.12.3 Version 0.19.0 (October 2, 2016)

This is a major release from 0.18.1 and includes number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `merge_asof()` for asof-style time-series joining, see here
- `.rolling()` is now time-series aware, see here
- `read_csv()` now supports parsing Categorical data, see here
- A function `union_categorical()` has been added for combining categoricals, see here
- `PeriodIndex` now has its own period dtype, and changed to be more consistent with other `Index` classes. See here
- Sparse data structures gained enhanced support of int and bool dtypes, see here
- Comparison operations with `Series` no longer ignores the index, see here for an overview of the API changes.
- Introduction of a pandas development API for utility functions, see here.
- Deprecation of `Panel4D` and `PanelND`. We recommend to represent these types of n-dimensional data with the `xarray` package.
- Removal of the previously deprecated modules `pandas.io.data`, `pandas.io.wb`, `pandas.tools.rplot`.

**Warning:** pandas >= 0.19.0 will no longer silence numpy ufunc warnings upon import, see here.

What's new in v0.19.0

- New features
  - Function `merge_asof` for asof-style time-series joining
  - Method `.rolling()` is now time-series aware
  - Method `read_csv()` has improved support for duplicate column names
  - Method `read_csv` supports parsing Categorical directly
  - Categorical concatenation
  - Semi-month offsets
  - New Index methods
  - Google BigQuery enhancements
  - Fine-grained NumPy errstate
  - Method `get_dummies` now returns integer dtypes
  - Downcast values to smallest possible dtype in `to_numeric`
- pandas development API
- Other enhancements

• API changes
  - Series.tolist() will now return Python types
  - Series operators for different indexes
    * Arithmetic operators
    * Comparison operators
    * Logical operators
    * Flexible comparison methods
  - Series type promotion on assignment
  - Function .to_datetime() changes
  - Merging changes
  - Method .describe() changes
  - Period changes
    * The PeriodIndex now has period dtype
    * Period(’NaT’) now returns pd.NaT
    * PeriodIndex.values now returns array of Period object
  - Index + / - no longer used for set operations
  - Index.difference and .symmetric_difference changes
  - Index.unique consistently returns Index
  - MultiIndex constructors, groupby and set_index preserve categorical dtypes
  - Function read_csv will progressively enumerate chunks
  - Sparse changes
    * Types int64 and bool support enhancements
    * Operators now preserve dtypes
    * Other sparse fixes
  - Indexer dtypes changes
  - Other API changes

• Deprecations
• Removal of prior version deprecations/changes
• Performance improvements
• Bug fixes
• Contributors
New features

Function `merge_asof` for asof-style time-series joining

A long-time requested feature has been added through the `merge_asof()` function, to support asof style joining of time-series (GH1870, GH13695, GH13709, GH13902). Full documentation is [here](#). The `merge_asof()` performs an asof merge, which is similar to a left-join except that we match on nearest key rather than equal keys.

```
In [1]: left = pd.DataFrame({'a': [1, 5, 10], 'left_val': ['a', 'b', 'c']})
In [2]: right = pd.DataFrame({'a': [1, 2, 3, 6, 7], 'right_val': [1, 2, 3, 6, 7]})
In [3]: left
Out[3]:
a  left_val
0   a
1   b
2   c

[3 rows x 2 columns]

In [4]: right
Out[4]:
a  right_val
0   1
1   2
2   3
3   6
4   7

[5 rows x 2 columns]
```

We typically want to match exactly when possible, and use the most recent value otherwise.

```
In [5]: pd.merge_asof(left, right, on="a")
Out[5]:
a  left_val  right_val
0   1       a  1
1   5       b  3
2  10       c  7

[3 rows x 3 columns]
```

We can also match rows ONLY with prior data, and not an exact match.

```
In [6]: pd.merge_asof(left, right, on="a", allow_exact_matches=False)
Out[6]:
a  left_val  right_val
0   1       a  NaN
1   5       b  3.0
2  10       c  7.0
```

(continues on next page)
In a typical time-series example, we have trades and quotes and we want to asof-join them. This also illustrates using the by parameter to group data before merging.

```python
In [7]: trades = pd.DataFrame(
    ...:     {  
    ...:         "time": pd.to_datetime(
    ...:             [  
    ...:                 "20160525 13:30:00.023",  
    ...:                 "20160525 13:30:00.038",  
    ...:                 "20160525 13:30:00.048",  
    ...:                 "20160525 13:30:00.048",  
    ...:                 "20160525 13:30:00.048",  
    ...:             ],  
    ...:         "ticker": ["MSFT", "MSFT", "GOOG", "GOOG", "AAPL"],  
    ...:         "price": [51.95, 51.95, 720.77, 720.92, 98.00],  
    ...:         "quantity": [75, 155, 100, 100, 100],  
    ...:     },  
    ...:     columns=["time", "ticker", "price", "quantity"],  
    ...: )

In [8]: quotes = pd.DataFrame(
    ...:     {  
    ...:         "time": pd.to_datetime(
    ...:             [  
    ...:                 "20160525 13:30:00.023",  
    ...:                 "20160525 13:30:00.023",  
    ...:                 "20160525 13:30:00.030",  
    ...:                 "20160525 13:30:00.041",  
    ...:                 "20160525 13:30:00.048",  
    ...:                 "20160525 13:30:00.049",  
    ...:                 "20160525 13:30:00.072",  
    ...:                 "20160525 13:30:00.075",  
    ...:             ],  
    ...:         "ticker": ["GOOG", "MSFT", "MSFT", "MSFT", "GOOG", "AAPL", "GOOG", "MSFT"],  
    ...:         "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.01],  
    ...:         "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.03],  
    ...:     },  
    ...:     columns=["time", "ticker", "bid", "ask"],  
    ...: )

In [9]: trades
Out[9]:
     time  ticker  price  quantity
0 2016-05-25 13:30:00.023 MSFT   51.95     75
```
An asof merge joins on the on, typically a datetimelike field, which is ordered, and in this case we are using a grouper in the by field. This is like a left-outer join, except that forward filling happens automatically taking the most recent non-NaN value.

Method .rolling() is now time-series aware

.rolling() objects are now time-series aware and can accept a time-series offset (or convertible) for the window argument (GH13327, GH12995). See the full documentation here.
This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

In [14]: dft.rolling(2).sum()

Out[14]:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 09:00:00</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-01 09:00:01</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:02</td>
<td>3.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:03</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-01 09:00:04</td>
<td>NaN</td>
</tr>
</tbody>
</table>

[5 rows x 1 columns]

In [15]: dft.rolling(2, min_periods=1).sum()

Out[15]:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 09:00:00</td>
<td>0.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:01</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:02</td>
<td>3.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:03</td>
<td>2.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:04</td>
<td>4.0</td>
</tr>
</tbody>
</table>

[5 rows x 1 columns]

Specifying an offset allows a more intuitive specification of the rolling frequency.

In [16]: dft.rolling("2s").sum()

Out[16]:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 09:00:00</td>
<td>0.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:01</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:02</td>
<td>3.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:03</td>
<td>2.0</td>
</tr>
<tr>
<td>2013-01-01 09:00:04</td>
<td>4.0</td>
</tr>
</tbody>
</table>

[5 rows x 1 columns]

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.
In [18]: dft
Out[18]:

    B
foo
2013-01-01 09:00:00    0.0
2013-01-01 09:00:02    1.0
2013-01-01 09:00:03    2.0
2013-01-01 09:00:05   NaN
2013-01-01 09:00:06    4.0

[5 rows x 1 columns]

In [19]: dft.rolling(2).sum()
Out[19]:

    B
foo
2013-01-01 09:00:00   NaN
2013-01-01 09:00:02    1.0
2013-01-01 09:00:03    3.0
2013-01-01 09:00:05   NaN
2013-01-01 09:00:06   NaN

[5 rows x 1 columns]

Using the time-specification generates variable windows for this sparse data.

In [20]: dft.rolling("2s").sum()
Out[20]:

    B
foo
2013-01-01 09:00:00    0.0
2013-01-01 09:00:02    1.0
2013-01-01 09:00:03    3.0
2013-01-01 09:00:05   NaN
2013-01-01 09:00:06    4.0

[5 rows x 1 columns]

Furthermore, we now allow an optional on parameter to specify a column (rather than the default of the index) in a DataFrame.
In [21]: dft = dft.reset_index()

In [22]: dft
Out[22]:
   foo  B
0  2013-01-01 09:00:00  0.0
1  2013-01-01 09:00:02  1.0
2  2013-01-01 09:00:03  2.0
3  2013-01-01 09:00:05  NaN
4  2013-01-01 09:00:06  4.0

[5 rows x 2 columns]

In [23]: dft.rolling("2s", on="foo").sum()
Out[23]:
   foo  B
0  2013-01-01 09:00:00  0.0
1  2013-01-01 09:00:02  1.0
2  2013-01-01 09:00:03  3.0
3  2013-01-01 09:00:05  NaN
4  2013-01-01 09:00:06  4.0

[5 rows x 2 columns]

Method read_csv has improved support for duplicate column names

Duplicate column names are now supported in read_csv() whether they are in the file or passed in as the names parameter (GH7160, GH9424)

In [24]: data = "0,1,2
   :3,4,5"

In [25]: names = ["a", "b", "a"]

Previous behavior:

In [2]: pd.read_csv(StringIO(data), names=names)
Out[2]:
   a  b  a
0  2  1  2
1  5  4  5

The first a column contained the same data as the second a column, when it should have contained the values [0, 3].

New behavior:

In [26]: pd.read_csv(StringIO(data), names=names)
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
Input In [26], in <cell line: 1>()
----> 1 pd.read_csv(StringIO(data), names=names)

File /pandas/pandas/util/_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>..__decorate.<locals>._wrappe
   311     *args, **kwargs)
(continues on next page)
if len(args) > num_allow_args:
    warnings.warn(
        msg.format(arguments=arguments),
        FutureWarning,
        stacklevel=stacklevel,
    )
-> return func(*args, **kwargs)

File /pandas/pandas/io/parsers/readers.py:680, in read_csv(filepath_or_buffer, sep, ...
   delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, ...
   engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, ...
   nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, ...
   infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates, iterator, ...
   chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, ...
   doublequote, escapechar, comment, encoding, encoding_errors, dialect, error_bad_lines, ...
   warn_bad_lines, on_bad_lines, delim_whitespace, low_memory, memory_map, float_ ... 
   precision, storage_options)
   kwds_defaults = _refine_defaults_read( 
       dialect, 
       delimiter,
       (...)
       defaults={"delimiter": ","},
   )
   kwds.update(kwds_defaults)
-> return _read(filepath_or_buffer, kwds)

File /pandas/pandas/io/parsers/readers.py:572, in _read(filepath_or_buffer, kwds)
   nrows = kwds.get("nrows", None)
   # Check for duplicates in names.
   -> _validate_names(kwds.get("names", None))
   # Create the parser.
   parser = TextFileReader(filepath_or_buffer, **kwds)

File /pandas/pandas/io/parsers/readers.py:532, in _validate_names(names)
   if names is not None:
   -> raise ValueError("Duplicate names are not allowed.")
   if not (is_list_like(names, allow_sets=False) or isinstance(names, abc.KeysView)):
   -> raise ValueError("Names should be an ordered collection.")

ValueError: Duplicate names are not allowed.
Method `read_csv` supports parsing `Categorical` directly

The `read_csv()` function now supports parsing a `Categorical` column when specified as a dtype (GH10153). Depending on the structure of the data, this can result in a faster parse time and lower memory usage compared to converting to `Categorical` after parsing. See the io docs here.

```python
In [27]: data = """"n
data=""""
    .....: col1,col2,col3
    .....: a,b,1
    .....: a,b,2
    .....: c,d,3
    .....: """
        n

In [28]: pd.read_csv(StringIO(data))
Out[28]:
   col1   col2   col3
0     a     b     1
1     a     b     2
2     c     d     3
[3 rows x 3 columns]

In [29]: pd.read_csv(StringIO(data)).dtypes
Out[29]:
    col1    object
    col2    object
    col3      int64
Length: 3, dtype: object

In [30]: pd.read_csv(StringIO(data), dtype="category").dtypes
Out[30]:
    col1    category
    col2    category
    col3    category
Length: 3, dtype: object

Individual columns can be parsed as a `Categorical` using a dict specification

```python
In [31]: pd.read_csv(StringIO(data), dtype={"col1": "category"}).dtypes
Out[31]:
    col1    category
    col2    object
    col3      int64
Length: 3, dtype: object
```

**Note:** The resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the `to_numeric()` function, or as appropriate, another converter such as `to_datetime()`.

```python
In [32]: df = pd.read_csv(StringIO(data), dtype="category")

In [33]: df.dtypes
```

(continues on next page)
Out[33]:
col1  category
      col2  category
      col3  category
Length: 3, dtype: object

In [34]: df['col3']
Out[34]:
0  1
1  2
2  3
Name: col3, Length: 3, dtype: category
Categories (3, object): ['1', '2', '3']

In [35]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [36]: df['col3']
Out[36]:
0  1
1  2
2  3
Name: col3, Length: 3, dtype: category
Categories (3, int64): [1, 2, 3]

**Categorical concatenation**

- A function `union_categoricals()` has been added for combining categoricals, see *Unioning Categoricals* (GH13361, GH13763, GH13846, GH14173)

    In [37]: from pandas.api.types import union_categoricals
    
    In [38]: a = pd.Categorical(['b', 'c'])
    
    In [39]: b = pd.Categorical(['a', 'b'])
    
    In [40]: union_categoricals([a, b])
    Out[40]:
    ['b', 'c', 'a', 'b']
    Categories (3, object): ['b', 'c', 'a']

- `concat` and `append` now can concat category dtypes with different categories as object dtype (GH13524)

    In [41]: s1 = pd.Series(['a', 'b'], dtype='category')
    
    In [42]: s2 = pd.Series(['b', 'c'], dtype='category')

**Previous behavior:**

    In [1]: pd.concat([s1, s2])
    ValueError: incompatible categories in categorical concat
New behavior:

```python
In [43]: pd.concat([s1, s2])
Out[43]:
   0  a
   1  b
   0  b
   1  c
Length: 4, dtype: object
```

Semi-month offsets

pandas has gained new frequency offsets, `SemiMonthEnd` (‘SM’) and `SemiMonthBegin` (‘SMS’). These provide date offsets anchored (by default) to the 15th and end of month, and 15th and 1st of month respectively. (GH1543)

```python
In [44]: from pandas.tseries.offsets import SemiMonthEnd, SemiMonthBegin
```

**SemiMonthEnd:**

```python
In [45]: pd.Timestamp("2016-01-01") + SemiMonthEnd()
Out[45]: Timestamp('2016-01-15 00:00:00')

In [46]: pd.date_range("2015-01-01", freq="SM", periods=4)
```

**SemiMonthBegin:**

```python
In [47]: pd.Timestamp("2016-01-01") + SemiMonthBegin()
Out[47]: Timestamp('2016-01-15 00:00:00')

In [48]: pd.date_range("2015-01-01", freq="SMS", periods=4)
Out[48]: DatetimeIndex(['2015-01-01', '2015-01-15', '2015-02-01', '2015-02-15'], dtype='datetime64[ns]', freq='SMS')
```

Using the anchoring suffix, you can also specify the day of month to use instead of the 15th.

```python
In [49]: pd.date_range("2015-01-01", freq="SMS-16", periods=4)
Out[49]: DatetimeIndex(['2015-01-01', '2015-01-16', '2015-02-01', '2015-02-16'], dtype='datetime64[ns]', freq='SMS')

In [50]: pd.date_range("2015-01-01", freq="SM-14", periods=4)
Out[50]: DatetimeIndex(['2015-01-14', '2015-01-31', '2015-02-14', '2015-02-28'], dtype='datetime64[ns]', freq='SM')
```
New Index methods

The following methods and options are added to Index, to be more consistent with the Series and DataFrame API. Index now supports the .where() function for same shape indexing (GH13170)

```
In [51]: idx = pd.Index(['a', 'b', 'c'])
...
In [52]: idx.where([True, False, True])
Out[52]: Index(['a', nan, 'c'], dtype='object')
```

Index now supports .dropna() to exclude missing values (GH6194)

```
In [53]: idx = pd.Index([1, 2, np.nan, 4])
...
In [54]: idx.dropna()
Out[54]: Float64Index([1.0, 2.0, 4.0], dtype='float64')
```

For MultiIndex, values are dropped if any level is missing by default. Specifying how='all' only drops values where all levels are missing.

```
In [55]: midx = pd.MultiIndex.from_arrays([[1, 2, np.nan, 4], [1, 2, np.nan, np.nan]])
...
In [56]: midx
Out[56]: MultiIndex([(1.0, 1.0),
                (2.0, 2.0),
                (nan, nan),
                (4.0, nan)],
               )
```

```
In [57]: midx.dropna()
Out[57]: MultiIndex([(1, 1),
                (2, 2)],
               )
```

```
In [58]: midx.dropna(how='all')
Out[58]: MultiIndex([(1, 1.0),
                (2, 2.0),
                (4, nan)],
               )
```

Index now supports .str.extractall() which returns a DataFrame, see the docs here (GH10008, GH13156)

```
In [59]: idx = pd.Index(['a1a2', 'b1', 'c1'])
...
In [60]: idx.str.extractall(r'\[ab\](?P<digit>\d)')
```

(continues on next page)
Index.astype() now accepts an optional boolean argument copy, which allows optional copying if the requirements on dtype are satisfied (GH13209)

**Google BigQuery enhancements**

- The `read_gbq()` method has gained the `dialect` argument to allow users to specify whether to use BigQuery’s legacy SQL or BigQuery’s standard SQL. See the docs for more details (GH13615).
- The `to_gbq()` method now allows the DataFrame column order to differ from the destination table schema (GH11359).

**Fine-grained NumPy errstate**

Previous versions of pandas would permanently silence numpy’s ufunc error handling when pandas was imported. pandas did this in order to silence the warnings that would arise from using numpy ufuncs on missing data, which are usually represented as NaNs. Unfortunately, this silenced legitimate warnings arising in non-pandas code in the application. Starting with 0.19.0, pandas will use the `numpy.errstate` context manager to silence these warnings in a more fine-grained manner, only around where these operations are actually used in the pandas code base. (GH13109, GH13145)

After upgrading pandas, you may see new `RuntimeWarnings` being issued from your code. These are likely legitimate, and the underlying cause likely existed in the code when using previous versions of pandas that simply silenced the warning. Use `numpy.errstate` around the source of the `RuntimeWarning` to control how these conditions are handled.

**Method get_dummies now returns integer dtypes**

The `pd.get_dummies` function now returns dummy-encoded columns as small integers, rather than floats (GH8725). This should provide an improved memory footprint.

**Previous behavior:**

```
In [1]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[1]:
a    float64
b    float64
c    float64
dtype: object
```

**New behavior:**

```
In [61]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[61]:
a    uint8
b    uint8
c    uint8
Length: 3, dtype: object
```
Downcast values to smallest possible dtype in `to_numeric`

`pd.to_numeric()` now accepts a `downcast` parameter, which will downcast the data if possible to smallest specified numerical dtype (GH13352)

```python
In [62]: s = ["1", 2, 3]
In [63]: pd.to_numeric(s, downcast="unsigned")
Out[63]: array([1, 2, 3], dtype=uint8)
In [64]: pd.to_numeric(s, downcast="integer")
Out[64]: array([1, 2, 3], dtype=int8)
```

**pandas development API**

As part of making pandas API more uniform and accessible in the future, we have created a standard sub-package of pandas, `pandas.api` to hold public API's. We are starting by exposing type introspection functions in `pandas.api.types`. More sub-packages and officially sanctioned API's will be published in future versions of pandas (GH13147, GH13634)

The following are now part of this API:

```python
In [65]: import pprint
In [66]: from pandas.api import types
In [67]: funcs = [f for f in dir(types) if not f.startswith("_")]
In [68]: pprint.pprint(funcs)
['CategoricalDtype',
 'DatetimeTZDtype',
 'IntervalDtype',
 'PeriodDtype',
 'infer_dtype',
 'is_array_like',
 'is_bool',
 'is_bool_dtype',
 'is_categorical',
 'is_categorical_dtype',
 'is_complex',
 'is_complex_dtype',
 'is_datetime64_any_dtype',
 'is_datetime64_dtype',
 'is_datetime64_ns_dtype',
 'is_datetime64tz_dtype',
 'is_dict_like',
 'is_dtype_equal',
 'is_extension_array_dtype',
 'is_extension_type',
 'is_file_like',
 'is_float',
 'is_float_dtype',
```

(continues on next page)
is_hashable',
'is_int64_dtype',
'is_integer',
'is_integer_dtype',
'is_interval',
'is_interval_dtype',
'is_iterator',
'is_list_like',
'is_named_tuple',
'is_number',
'is_numeric_dtype',
'is_object_dtype',
'is_period_dtype',
're',
're_compilable',
'scalar',
'signed_integer_dtype',
'sparse',
'string_dtype',
'timedeltas64_dtype',
'timedeltas64_ns_dtype',
'unsigned_integer_dtype',
'version',
'union_categoricals']

Note: Calling these functions from the internal module pandas.core.common will now show a
DeprecationWarning (GH13990)

Other enhancements

• Timestamp can now accept positional and keyword parameters similar to datetime.datetime() (GH10758,
GH11630)

In [69]: pd.Timestamp(2012, 1, 1)
Out[69]: Timestamp('2012-01-01 00:00:00')

In [70]: pd.Timestamp(year=2012, month=1, day=1, hour=8, minute=30)
Out[70]: Timestamp('2012-01-01 08:30:00')

• The .resample() function now accepts a on= or level= parameter for resampling on a datetimelike column
or MultiIndex level (GH13500)

In [71]: df = pd.DataFrame(
       ....:   {"date": pd.date_range("2015-01-01", freq="W", periods=5), "a": np.
       ....:     arange(5)},
       ....:   index=pd.MultiIndex.from_arrays(
       ....:     [[1, 2, 3, 4, 5], pd.date_range("2015-01-01", freq="W",
       ....:     periods=5)],
       ....:     names=["v", "d"],
       ....:   )
       ....:   )
In [72]: df
def

Out[72]:

<table>
<thead>
<tr>
<th></th>
<th>date</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2015-01-04</td>
<td>2015-01-04</td>
</tr>
<tr>
<td>2</td>
<td>2015-01-11</td>
<td>2015-01-11</td>
</tr>
<tr>
<td>3</td>
<td>2015-01-18</td>
<td>2015-01-18</td>
</tr>
<tr>
<td>4</td>
<td>2015-01-25</td>
<td>2015-01-25</td>
</tr>
<tr>
<td>5</td>
<td>2015-02-01</td>
<td>2015-02-01</td>
</tr>
</tbody>
</table>

[5 rows x 2 columns]

In [73]: df.resample("M", on="date").sum()

Out[73]:

<table>
<thead>
<tr>
<th>date</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-31</td>
<td>6</td>
</tr>
<tr>
<td>2015-02-28</td>
<td>4</td>
</tr>
</tbody>
</table>

[2 rows x 1 columns]

In [74]: df.resample("M", level="d").sum()

Out[74]:

<table>
<thead>
<tr>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-31</td>
</tr>
<tr>
<td>2015-02-28</td>
</tr>
</tbody>
</table>

[2 rows x 1 columns]

- The `get_credentials()` method of `GbqConnector` can now first try to fetch the application default credentials. See the docs for more details (GH13577).

- The `.tz_localize()` method of `DatetimeIndex` and `Timestamp` has gained the `errors` keyword, so you can potentially coerce nonexistent timestamps to NaT. The default behavior remains to raising a `NonExistentTimeError` (GH13057).

- `.to_hdf/read_hdf()` now accept path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path (GH11773).

- The `pd.read_csv()` with engine='python' has gained support for the `decimal` (GH12933), `na_filter` (GH13321) and the `memory_map` option (GH13381).

- Consistent with the Python API, `pd.read_csv()` will now interpret `+inf` as positive infinity (GH13274).

- The `pd.read_html()` has gained support for the `na_values`, `converters`, `keep_default_na` options (GH13461).

- `Categorical.astype()` now accepts an optional boolean argument `copy`, effective when dtype is categorical (GH13209).

- `DataFrame` has gained the `.asof()` method to return the last non-NaN values according to the selected subset.
• The DataFrame constructor will now respect key ordering if a list of OrderedDict objects are passed in (GH13304)

• pd.read_html() has gained support for the decimal option (GH12907)

• Series has gained the properties .is_monotonic, .is_monotonic_increasing, .is_monotonic_decreasing, similar to Index (GH13336)

• DataFrame.to_sql() now allows a single value as the SQL type for all columns (GH11886).

• Series.append now supports the ignore_index option (GH13677)

• .to_stata() and StataWriter can now write variable labels to Stata dta files using a dictionary to make column names to labels (GH13535, GH13536)

• .to_stata() and StataWriter will automatically convert datetime64[ns] columns to Stata format %tc, rather than raising a ValueError (GH12259)

• read_stata() and StataReader raise with a more explicit error message when reading Stata files with repeated value labels when convert_categoricals=True (GH13923)

• DataFrame.style will now render sparsified MultiIndexes (GH11655)

• DataFrame.style will now show column level names (e.g. DataFrame.columns.names) (GH13775)

• DataFrame has gained support to re-order the columns based on the values in a row using df.sort_values(by='...', axis=1) (GH10806)

```
In [75]: df = pd.DataFrame({"A": [2, 7], "B": [3, 5], "C": [4, 8]}, index=["row1", "row2"])

In [76]: df
Out[76]:
     A  B  C
row1 2  3  4
row2 7  5  8
[2 rows x 3 columns]

In [77]: df.sort_values(by="row2", axis=1)
Out[77]:
     B  A  C
row1 3  2  4
row2 5  7  8
[2 rows x 3 columns]
```

• Added documentation to I/O regarding the perils of reading in columns with mixed dtypes and how to handle it (GH13746)

• to_html() now has a border argument to control the value in the opening <table> tag. The default is the value of the html.border option, which defaults to 1. This also affects the notebook HTML repr, but since Jupyter’s CSS includes a border-width attribute, the visual effect is the same. (GH11563).

• Raise ImportError in the sql functions when sqlalchemy is not installed and a connection string is used (GH11920).

• Compatibility with matplotlib 2.0. Older versions of pandas should also work with matplotlib 2.0 (GH13333)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Timestamp, Period, DatetimeIndex, PeriodIndex and .dt accessor have gained a .is_leap_year property to check whether the date belongs to a leap year. (GH13727)
- astype() will now accept a dict of column name to data types mapping as the dtype argument. (GH12086)
- The pd.read_json and DataFrame.to_json has gained support for reading and writing json lines with lines option see Line delimited json (GH9180)
- read_excel() now supports the true_values and false_values keyword arguments (GH13347)
- groupby() will now accept a scalar and a single-element list for specifying level on a non-MultiIndex grouper. (GH13907)
- Non-convertible dates in an excel date column will be returned without conversion and the column will be object dtype, rather than raising an exception (GH10001).
- pd.Timedelta(None) is now accepted and will return NaT, mirroring pd.Timestamp (GH13687)
- pd.read_stata() can now handle some format 111 files, which are produced by SAS when generating Stata dta files (GH11526)
- Series and Index now support divmod which will return a tuple of series or indices. This behaves like a standard binary operator with regards to broadcasting rules (GH14208).

API changes

**Series.tolist() will now return Python types**

Series.tolist() will now return Python types in the output, mimicking NumPy .tolist() behavior (GH10904)

```
In [78]: s = pd.Series([1, 2, 3])
```

**Previous behavior:**

```
In [7]: type(s.tolist()[0])
Out[7]:
<class 'numpy.int64'>
```

**New behavior:**

```
In [79]: type(s.tolist()[0])
Out[79]: int
```

**Series operators for different indexes**

Following Series operators have been changed to make all operators consistent, including DataFrame (GH1134, GH4581, GH13538)

- Series comparison operators now raise ValueError when index are different.
- Series logical operators align both index of left and right hand side.

**Warning:** Until 0.18.1, comparing Series with the same length, would succeed even if the .index are different (the result ignores .index). As of 0.19.0, this will raises ValueError to be more strict. This section also describes how to keep previous behavior or align different indexes, using the flexible comparison methods like .eq.
As a result, Series and DataFrame operators behave as below:

### Arithmetic operators

Arithmetic operators align both index (no changes).

```
In [80]: s1 = pd.Series([1, 2, 3], index=list("ABC"))
In [81]: s2 = pd.Series([2, 2, 2], index=list("ABD"))
In [82]: s1 + s2
Out[82]:
A   3.0
B   4.0
C  NaN
D  NaN
Length: 4, dtype: float64
```

```
In [83]: df1 = pd.DataFrame([1, 2, 3], index=list("ABC"))
In [84]: df2 = pd.DataFrame([2, 2, 2], index=list("ABD"))
In [85]: df1 + df2
Out[85]:
     0
A  3.0
B  4.0
C  NaN
D  NaN
[4 rows x 1 columns]
```

### Comparison operators

Comparison operators raise ValueError when .index are different.

Previous behavior (Series):

Series compared values ignoring the .index as long as both had the same length:

```
In [1]: s1 == s2
Out[1]:
A  False
B   True
C  False
dtype: bool
```

New behavior (Series):

```
In [2]: s1 == s2
Out[2]:
ValueError: Can only compare identically-labeled Series objects
```
Note: To achieve the same result as previous versions (compare values based on locations ignoring `.index`), compare both `.values`.

In [86]: s1.values == s2.values  
Out[86]: array([False, True, False])

If you want to compare `Series` aligning its `.index`, see flexible comparison methods section below:

In [87]: s1.eq(s2)  
Out[87]:
A  True
B  False
C  False
D  False
Length: 4, dtype: bool

Current behavior (DataFrame, no change):

In [3]: df1 == df2  
Out[3]:
ValueError: Can only compare identically-labeled DataFrame objects

Logical operators

Logical operators align both `.index` of left and right hand side.

Previous behavior (Series), only left hand side index was kept:

In [4]: s1 = pd.Series([True, False, True], index=list('ABC'))  
In [5]: s2 = pd.Series([True, True, True], index=list('ABD'))  
In [6]: s1 & s2  
Out[6]:
A   True
B   False
C   False
dtype: bool

New behavior (Series):

In [88]: s1 = pd.Series([True, False, True], index=list('ABC'))  
In [89]: s2 = pd.Series([True, True, True], index=list('ABD'))  
In [90]: s1 & s2  
Out[90]:
A   True
B   False
C   False
D   False
Length: 4, dtype: bool
**Note:** Series logical operators fill a NaN result with False.

**Note:** To achieve the same result as previous versions (compare values based on only left hand side index), you can use `reindex_like`:

```python
In [91]: s1 & s2.reindex_like(s1)
Out[91]:
A   True
B    False
C    False
Length: 3, dtype: bool
```

**Current behavior** (DataFrame, no change):

```python
In [92]: df1 = pd.DataFrame([True, False, True], index=list("ABC"))
In [93]: df2 = pd.DataFrame([True, True, True], index=list("ABD"))
In [94]: df1 & df2
Out[94]:
   0
A  True
B    False
C    False
D    False
[4 rows x 1 columns]
```

**Flexible comparison methods**

Series flexible comparison methods like `eq`, `ne`, `le`, `lt`, `ge` and `gt` now align both index. Use these operators if you want to compare two Series which has the different index.

```python
In [95]: s1 = pd.Series([1, 2, 3], index=list("a", "b", "c"))
In [96]: s2 = pd.Series([2, 2, 2], index=list("b", "c", "d"))
In [97]: s1.eq(s2)
Out[97]:
a    False
b    True
c    False
d    False
Length: 4, dtype: bool
In [98]: s1.ge(s2)
Out[98]:
a     False
b     True
```
Previously, this worked the same as comparison operators (see above).

**Series type promotion on assignment**

A Series will now correctly promote its dtype for assignment with incompat values to the current dtype (GH13234)

```python
In [99]: s = pd.Series()

Previous behavior:

```python
In [2]: s["a"] = pd.Timestamp("2016-01-01")
```

```python
In [3]: s["b"] = 3.0
TypeError: invalid type promotion
```

New behavior:

```python
In [100]: s["a"] = pd.Timestamp("2016-01-01")
In [101]: s["b"] = 3.0
```

```python
In [102]: s
Out[102]:
   a  2016-01-01
   b   3.0
Length: 2, dtype: object
```

```python
In [103]: s.dtype
Out[103]: dtype('O')
```

**Function .to_datetime() changes**

Previously if .to_datetime() encountered mixed integers/floats and strings, but no datetimes with errors='coerce' it would convert all to NaT.

**Previous behavior:**

```python
In [2]: pd.to_datetime([1, 'foo'], errors='coerce')
Out[2]: DatetimeIndex(['NaT', 'NaT'], dtype='datetime64[ns]', freq=None)
```

**Current behavior:**

This will now convert integers/floats with the default unit of ns.

```python
In [104]: pd.to_datetime([1, "foo"], errors="coerce")
Out[104]: DatetimeIndex(['1970-01-01 00:00:00.000000001', 'NaT'], dtype='datetime64[ns]', freq=None)
```
Bug fixes related to \texttt{.to_datetime()}:

- Bug in \texttt{pd.to_datetime()} when passing integers or floats, and no unit and \texttt{errors='coerce'} (GH13180).
- Bug in \texttt{pd.to_datetime()} when passing invalid data types (e.g. bool); will now respect the \texttt{errors} keyword (GH13176)
- Bug in \texttt{pd.to_datetime()} which overflowed on int8, and int16 dtypes (GH13451)
- Bug in \texttt{pd.to_datetime()} raise \texttt{AttributeError} with NaN and the other string is not valid when \texttt{errors='ignore'} (GH12424)
- Bug in \texttt{pd.to_datetime()} did not cast floats correctly when unit was specified, resulting in truncated datetime (GH13834)

**Merging changes**

Merging will now preserve the dtype of the join keys (GH8596)

\begin{Verbatim}
\textbf{In [105]}: df1 = pd.DataFrame({"key": [1], "v1": [10]})

\textbf{In [106]}: df1
\textbf{Out[106]}:
  key  v1
  0   1  10

[1 rows x 2 columns]

\textbf{In [107]}: df2 = pd.DataFrame({"key": [1, 2], "v1": [20, 30]})

\textbf{In [108]}: df2
\textbf{Out[108]}:
  key  v1
  0   1  20
  1   2  30

[2 rows x 2 columns]
\end{Verbatim}

**Previous behavior:**

\begin{Verbatim}
\textbf{In [5]}: pd.merge(df1, df2, how='outer')
\textbf{Out[5]}:
  key  v1
  0  1.0  10.0
  1  1.0  20.0
  2  2.0  30.0

\textbf{In [6]}: pd.merge(df1, df2, how='outer').dtypes
\textbf{Out[6]}:
  key   float64
  v1    float64
  dtype: object
\end{Verbatim}

**New behavior:**

We are able to preserve the join keys
### Example Code

```python
In [109]: pd.merge(df1, df2, how="outer")
Out[109]:
   key  v1
0   1   10
1   1   20
2   2   30

[3 rows x 2 columns]

In [110]: pd.merge(df1, df2, how="outer").dtypes
Out[110]:
   key  int64
   v1   int64

Length: 2, dtype: object
```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast, which is unchanged from previous.

```python
In [111]: pd.merge(df1, df2, how="outer", on="key")
Out[111]:
   key  v1_x  v1_y
0   1    10.0  20
1   2     NaN   30

[2 rows x 3 columns]

In [112]: pd.merge(df1, df2, how="outer", on="key").dtypes
Out[112]:
   key  int64
   v1_x float64
   v1_y  int64

Length: 3, dtype: object
```

### Method `.describe()` changes

Percentile identifiers in the index of a `.describe()` output will now be rounded to the least precision that keeps them distinct (GH13104)

```python
In [13]: s = pd.Series([0, 1, 2, 3, 4])

In [14]: df = pd.DataFrame([0, 1, 2, 3, 4])
```

**Previous behavior:**
The percentiles were rounded to at most one decimal place, which could raise `ValueError` for a data frame if the percentiles were duplicated.

```python
In [3]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[3]:
   percentiles
count 5.000000
mean 2.000000
std 1.581139
```

(continues on next page)
```python
In [4]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[4]:
...  
ValueError: cannot reindex from a duplicate axis

New behavior:

In [115]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[115]:
  count      5.000000
  mean        2.000000
  std        1.581139
  min        0.000080
  0.01%      0.000040
  0.05%      0.000200
  0.1%       0.004000
  50%        2.000000
  99.9%      3.996000
  99.95%     3.998000
  99.99%     3.999600
  max        4.000000
Length: 12, dtype: float64

In [116]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[116]:
    0
  count  5.000000
  mean  2.000000
  std  1.581139
  min  0.000000
  0.01%  0.000000
  0.05%  0.000000
  0.1%  0.000000
  50%  2.000000
  99.9%  3.996000
  99.95%  3.998000
  99.99%  3.999600
  max  4.000000

[12 rows x 1 columns]

Furthermore:
```
• Passing duplicated percentiles will now raise a `ValueError`.
• Bug in `.describe()` on a DataFrame with a mixed-dtype column index, which would previously raise a `TypeError` (GH13288)

**Period changes**

The PeriodIndex now has period dtype

PeriodIndex now has its own period dtype. The period dtype is a pandas extension dtype like category or the timezone aware dtype (datetime64[ns, tz]) (GH13941). As a consequence of this change, PeriodIndex no longer has an integer dtype:

**Previous behavior:**

```python
In [1]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')
In [2]: pi
Out[2]: PeriodIndex(['2016-08-01'], dtype='int64', freq='D')
In [3]: pd.api.types.is_integer_dtype(pi)
Out[3]: True
In [4]: pi.dtype
Out[4]: dtype('int64')
```

**New behavior:**

```python
In [117]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')
In [118]: pi
Out[118]: PeriodIndex(['2016-08-01'], dtype='period[D]')
In [119]: pd.api.types.is_integer_dtype(pi)
Out[119]: False
In [120]: pd.api.types.is_period_dtype(pi)
Out[120]: True
In [121]: pi.dtype
Out[121]: period[D]
In [122]: type(pi.dtype)
Out[122]: pandas.core.dtypes.dtypes.PeriodDtype
```
Period('NaT') now returns pd.NaT

Previously, Period has its own Period('NaT') representation different from pd.NaT. Now Period('NaT') has been changed to return pd.NaT. (GH12759, GH13582)

Previous behavior:

```python
In [5]: pd.Period('NaT', freq='D')
Out[5]: Period('NaT', 'D')
```

New behavior:

These result in pd.NaT without providing freq option.

```python
In [123]: pd.Period("NaT")
Out[123]: NaT
In [124]: pd.Period(None)
Out[124]: NaT
```

To be compatible with Period addition and subtraction, pd.NaT now supports addition and subtraction with int. Previously it raised ValueError.

Previous behavior:

```python
In [5]: pd.NaT + 1
...:
ValueError: Cannot add integral value to Timestamp without freq.
```

New behavior:

```python
In [125]: pd.NaT + 1
Out[125]: NaT
In [126]: pd.NaT - 1
Out[126]: NaT
```

PeriodIndex.values now returns array of Period object

.values is changed to return an array of Period objects, rather than an array of integers (GH13988).

Previous behavior:

```python
In [6]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')
In [7]: pi.values
Out[7]: array([492, 493])
```

New behavior:

```python
In [127]: pi = pd.PeriodIndex(["2011-01", "2011-02"], freq="M")
In [128]: pi.values
Out[128]: array([Period('2011-01', 'M'), Period('2011-02', 'M')], dtype=object)
```
Index + / - no longer used for set operations

Addition and subtraction of the base Index type and of DatetimeIndex (not the numeric index types) previously performed set operations (set union and difference). This behavior was already deprecated since 0.15.0 (in favor using the specific union() and difference() methods), and is now disabled. When possible, + and - are now used for element-wise operations, for example for concatenating strings or subtracting datetimes (GH8227, GH14127).

Previous behavior:

```python
In [1]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
FutureWarning: using '+' to provide set union with Indexes is deprecated, use '|' or .union()
Out[1]: Index(['a', 'b', 'c'], dtype='object')
```

New behavior: the same operation will now perform element-wise addition:

```python
In [129]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
Out[129]: Index(['aa', 'bc'], dtype='object')
```

Note that numeric Index objects already performed element-wise operations. For example, the behavior of adding two integer Indexes is unchanged. The base Index is now made consistent with this behavior.

```python
In [130]: pd.Index([1, 2, 3]) + pd.Index([2, 3, 4])
Out[130]: Int64Index([3, 5, 7], dtype='int64')
```

Further, because of this change, it is now possible to subtract two DatetimeIndex objects resulting in a TimedeltaIndex:

Previous behavior:

```python
In [1]: (pd.DatetimeIndex(['2016-01-01', '2016-01-02'])
   ...: - pd.DatetimeIndex(['2016-01-02', '2016-01-03']))
FutureWarning: using '-' to provide set differences with datetimelike Indexes is deprecated, use .difference()
Out[1]: DatetimeIndex(['2016-01-01'], dtype='datetime64[ns]', freq=None)
```

New behavior:

```python
In [131]: (pd.DatetimeIndex(['2016-01-01', '2016-01-02'])
   ...: - pd.DatetimeIndex(['2016-01-02', '2016-01-03']))
Out[131]: TimedeltaIndex(['-1 days', '-1 days'], dtype='timedelta64[ns]', freq=None)
```

Index.difference and .symmetric_difference changes

Index.difference and Index.symmetric_difference will now, more consistently, treat NaN values as any other values. (GH13514)

```python
In [132]: idx1 = pd.Index([1, 2, 3, np.nan])
In [133]: idx2 = pd.Index([0, 1, np.nan])
```

Previous behavior:
<table>
<thead>
<tr>
<th>In</th>
<th>idx1.difference(idx2)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>Float64Index([nan, 2.0, 3.0], dtype='float64')</td>
<td>[3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>idx1.symmetric_difference(idx2)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4]</td>
<td>Float64Index([0.0, nan, 2.0, 3.0], dtype='float64')</td>
<td>[4]</td>
</tr>
</tbody>
</table>

**New behavior:**

<table>
<thead>
<tr>
<th>In</th>
<th>idx1.difference(idx2)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[134]</td>
<td>Float64Index([2.0, 3.0], dtype='float64')</td>
<td>[134]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>idx1.symmetric_difference(idx2)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[135]</td>
<td>Float64Index([0.0, 2.0, 3.0], dtype='float64')</td>
<td>[135]</td>
</tr>
</tbody>
</table>

**Index.unique consistently returns Index**

Index.unique() now returns unique values as an Index of the appropriate dtype. (GH13395). Previously, most Index classes returned np.ndarray, and DatetimeIndex, TimedeltaIndex and PeriodIndex returned Index to keep metadata like timezone.

**Previous behavior:**

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Index([1, 2, 3]).unique()</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>array([1, 2, 3])</td>
<td>[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo').unique()</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00', '2011-01-03 00:00:00+09:00'], dtype='datetime64[ns, Asia/Tokyo]', freq=None)</td>
<td>[2]</td>
</tr>
</tbody>
</table>

**New behavior:**

<table>
<thead>
<tr>
<th>In</th>
<th>pd.Index([1, 2, 3]).unique()</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[136]</td>
<td>Int64Index([1, 2, 3], dtype='int64')</td>
<td>[136]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo').unique()</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[137]</td>
<td>DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00', '2011-01-03 00:00:00+09:00'], dtype='datetime64[ns, Asia/Tokyo]', freq=None)</td>
<td>[137]</td>
</tr>
</tbody>
</table>
**MultiIndex constructors, groupby and set_index preserve categorical dtypes**

MultiIndex.from_arrays and MultiIndex.from_product will now preserve categorical dtype in MultiIndex levels (GH13743, GH13854).

```
In [138]: cat = pd.Categorical(['a', 'b'], categories=list('bac'))
In [139]: lvl1 = ['foo', 'bar']
In [140]: midx = pd.MultiIndex.from_arrays([cat, lvl1])
In [141]: midx
Out[141]: MultiIndex([('a', 'foo'), ('b', 'bar')],

Previous behavior:

```
In [4]: midx.levels[0]
Out[4]: Index(['b', 'a', 'c'], dtype='object')
In [5]: midx.get_level_values(0)
Out[5]: Index(['a', 'b'], dtype='object')
```

**New behavior:** the single level is now a CategoricalIndex:

```
In [142]: midx.levels[0]
Out[142]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False, dtype='category')
In [143]: midx.get_level_values(0)
Out[143]: CategoricalIndex(['a', 'b'], categories=['b', 'a', 'c'], ordered=False, dtype='category')
```

An analogous change has been made to MultiIndex.from_product. As a consequence, groupby and set_index also preserve categorical dtypes in indexes

```
In [144]: df = pd.DataFrame({'A': [0, 1], 'B': [10, 11], 'C': cat})
In [145]: df_grouped = df.groupby(by=['A', 'C']).first()
In [146]: df_set_idx = df.set_index(['A', 'C'])
```

Previous behavior:

```
In [11]: df_grouped.index.levels[1]
Out[11]: Index(['b', 'a', 'c'], dtype='object', name='C')
In [12]: df_grouped.reset_index().dtypes
Out[12]:
   A     int64
   B   float64
   C    object
```

(continues on next page)
In [13]: df_set_idx.index.levels[1]
Out[13]: Index(['b', 'a', 'c'], dtype='object', name='C')
In [14]: df_set_idx.reset_index().dtypes
Out[14]:
   A    int64
   C  object
   B    int64
dtype: object

New behavior:

In [147]: df_grouped.index.levels[1]
Out[147]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
    dtype='category', name='C')
In [148]: df_grouped.reset_index().dtypes
Out[148]:
   A    int64
   C    category
   B  float64
Length: 3, dtype: object
In [149]: df_set_idx.index.levels[1]
Out[149]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
    dtype='category', name='C')
In [150]: df_set_idx.reset_index().dtypes
Out[150]:
   A    int64
   C    category
   B    int64
Length: 3, dtype: object

Function read_csv will progressively enumerate chunks

When read_csv() is called with chunksize=n and without specifying an index, each chunk used to have an independently generated index from 0 to n-1. They are now given instead a progressive index, starting from 0 for the first chunk, from n for the second, and so on, so that, when concatenated, they are identical to the result of calling read_csv() without the chunksize= argument (GH12185).

In [151]: data = "A,B\n0,1\n2,3\n4,5\n6,7"

Previous behavior:

In [2]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[2]:
   A  B
0  0  1
1  2  3

(continues on next page)
New behavior:

```python
In [152]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[152]:
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
[4 rows x 2 columns]
```

Sparse changes

These changes allow pandas to handle sparse data with more dtypes, and for work to make a smoother experience with data handling.

**Types int64 and bool support enhancements**

Sparse data structures now gained enhanced support of `int64` and `bool` dtype (GH667, GH13849).

Previously, sparse data were `float64` dtype by default, even if all inputs were of `int` or `bool` dtype. You had to specify `dtype` explicitly to create sparse data with `int64` dtype. Also, `fill_value` had to be specified explicitly because the default was `np.nan` which doesn't appear in `int64` or `bool` data.

```python
In [1]: pd.SparseArray([1, 2, 0, 0])
Out[1]:
[1.0, 2.0, 0.0, 0.0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

# specifying int64 dtype, but all values are stored in sp_values because
# fill_value default is np.nan
In [2]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[2]:
[1, 2, 0, 0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

In [3]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64, fill_value=0)
Out[3]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)
```
As of v0.19.0, sparse data keeps the input dtype, and uses more appropriate fill_value defaults (0 for int64 dtype, False for bool dtype).

```python
In [153]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[153]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)
```

```python
In [154]: pd.SparseArray([True, False, False, False])
Out[154]:
[True, False, False, False]
Fill: False
IntIndex
Indices: array([0], dtype=int32)
```

See the docs for more details.

**Operators now preserve dtypes**

- Sparse data structure now can preserve dtype after arithmetic ops (GH13848)

```python
s = pd.SparseSeries([0, 2, 0, 1], fill_value=0, dtype=np.int64)
s.dtype
s + 1
```

- Sparse data structure now support astype to convert internal dtype (GH13900)

```python
s = pd.SparseSeries([1.0, 0.0, 2.0, 0.0], fill_value=0)
s
s.astype(np.int64)
```

astype fails if data contains values which cannot be converted to specified dtype. Note that the limitation is applied to fill_value which default is np.nan.

```python
In [7]: pd.SparseSeries([1., np.nan, 2., np.nan]).astype(np.int64)
Out[7]:
ValueError: unable to coerce current fill_value nan to int64 dtype
```

**Other sparse fixes**

- Subclassed SparseDataFrame and SparseSeries now preserve class types when slicing or transposing. (GH13787)
- SparseArray with bool dtype now supports logical (bool) operators (GH14000)
- Bug in SparseSeries with MultiIndex [] indexing may raise IndexError (GH13144)
- Bug in SparseSeries with MultiIndex [] indexing result may have normal Index (GH13144)
- Bug in SparseDataFrame in which axis=None did not default to axis=0 (GH13048)
- Bug in SparseSeries and SparseDataFrame creation with object dtype may raise TypeError (GH11633)
Indexer dtype changes

Note: This change only affects 64 bit python running on Windows, and only affects relatively advanced indexing operations

Methods such as Index.get_indexer that return an indexer array, coerce that array to a “platform int”, so that it can be directly used in 3rd party library operations like numpy.take. Previously, a platform int was defined as np.int_ which corresponds to a C integer, but the correct type, and what is being used now, is np.intp, which corresponds to the C integer size that can hold a pointer (GH3033, GH13972).

These types are the same on many platform, but for 64 bit python on Windows, np.int_ is 32 bits, and np.intp is 64 bits. Changing this behavior improves performance for many operations on that platform.

Previous behavior:

```python
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
dtype('int32')
```

New behavior:

```python
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
dtype('int64')
```
Other API changes

- `Timestamp.to_pydatetime` will issue a `UserWarning` when `warn=True`, and the instance has a non-zero number of nanoseconds, previously this would print a message to stdout (GH14101).
- `Series.unique()` with datetime and timezone now returns return array of `Timestamp` with timezone (GH13565).
- `Panel.to_sparse()` will raise a `NotImplementedError` exception when called (GH13778).
- `Index.reshape()` will raise a `NotImplementedError` exception when called (GH12882).
- `.filter()` enforces mutual exclusion of the key arguments (GH12399).
- eval’s upcasting rules for `float32` types have been updated to be more consistent with NumPy’s rules. New behavior will not upcast to `float64` if you multiply a pandas `float32` object by a scalar `float64` (GH12388).
- An `UnsupportedFunctionCall` error is now raised if NumPy ufuncs like `np.mean` are called on groupby or resample objects (GH12811).
- `__setitem__` will no longer apply a callable rhs as a function instead of storing it. Call where directly to get the previous behavior (GH13299).
- Calls to .sample() will respect the random seed set via `numpy.random.seed(n)` (GH13161)
- Styler.apply is now more strict about the outputs your function must return. For axis=0 or axis=1, the output shape must be identical. For axis=None, the output must be a DataFrame with identical columns and index labels (GH13222).
- Float64Index.astype(int) will now raise ValueError if Float64Index contains NaN values (GH13149)
- TimedeltaIndex.astype(int) and DatetimeIndex.astype(int) will now return Int64Index instead of `np.array` (GH13209)
- Passing Period with multiple frequencies to normal Index now returns Index with object dtype (GH13664)
- PeriodIndex.fillna with Period has different freq now coerces to object dtype (GH13664)
- Faceted boxplots from DataFrame.boxplot(by=col) now return a Series when return_type is not None. Previously these returned an OrderedDict. Note that when return_type=None, the default, these still return a 2-D NumPy array (GH12216, GH7096).
- `pd.read_hdf` will now raise a ValueError instead of KeyError, if a mode other than r, r+ and a is supplied. (GH13623)
- `pd.read_csv()`, `pd.read_table()`, and `pd.read_hdf()` raise the builtin `FileNotFoundError` exception for Python 3.x when called on a nonexistent file; this is back-ported as `IOError` in Python 2.x (GH14086)
- More informative exceptions are passed through the csv parser. The exception type would now be the original exception type instead of `CFParserError` (GH13652).
- `pd.read_csv()` in the C engine will now issue a `ParserWarning` or raise a `ValueError` when `sep` encoded is more than one character long (GH14065)
- `DataFrame.values` will now return float64 with a DataFrame of mixed `int64` and `uint64` dtypes, conforming to `np.find_common_type` (GH10364, GH13917)
- `.groupby().groups` will now return a dictionary of `Index` objects, rather than a dictionary of `np.ndarray` or `lists` (GH14293)
pandas: powerful Python data analysis toolkit, Release 1.4.2

Deprecations

- `Series.reshape` and `Categorical.reshape` have been deprecated and will be removed in a subsequent release (GH12882, GH12882)
- `PeriodIndex.to_datetime` has been deprecated in favor of `PeriodIndex.to_timestamp` (GH8254)
- `Timestamp.to_datetime` has been deprecated in favor of `Timestamp.to_pydatetime` (GH8254)
- `Index.to_datetime` and `DatetimeIndex.to_datetime` have been deprecated in favor of `pd.to_datetime` (GH8254)
- `pandas.core.datetools` module has been deprecated and will be removed in a subsequent release (GH14094)
- `SparseList` has been deprecated and will be removed in a future version (GH13784)
- `DataFrame.to_html()` and `DataFrame.to_latex()` have dropped the `colSpace` parameter in favor of `col_space` (GH13857)
- `DataFrame.to_sql()` has deprecated the `flavor` parameter, as it is superfluous when SQLAlchemy is not installed (GH13611)
- Deprecated read_csv keywords:
  - `compact_ints` and `use_unsigned` have been deprecated and will be removed in a future version (GH13320)
  - `buffer_lines` has been deprecated and will be removed in a future version (GH13360)
  - `as_recarray` has been deprecated and will be removed in a future version (GH13373)
  - `skip_footer` has been deprecated in favor of `skipfooter` and will be removed in a future version (GH13349)
- top-level `pd.ordered_merge()` has been renamed to `pd.merge_ordered()` and the original name will be removed in a future version (GH13358)
- `Timestamp.offset` property (and named arg in the constructor), has been deprecated in favor of `freq` (GH12160)
- `pd.tseries.util.pivot_annual` is deprecated. Use `pivot_table` as alternative, an example is `here` (GH736)
- `pd.tseries.util.isleapyear` has been deprecated and will be removed in a subsequent release. Datetime-like now have a `.is_leap_year` property (GH13727)
- `Panel4D` and `PanelND` constructors are deprecated and will be removed in a future version. The recommended way to represent these types of n-dimensional data are with the `xarray` package. pandas provides a `to_xarray()` method to automate this conversion (GH13564).
- `pandas.tseries.frequencies.get_standard_freq` is deprecated. Use `pandas.tseries.frequencies.to_offset(freq).rule_code` instead (GH13874)
- `pandas.tseries.frequencies.to_offset`'s `freqstr` keyword is deprecated in favor of `freq` (GH13874)
- `Categorical.from_array` has been deprecated and will be removed in a future version (GH13854)
Removal of prior version deprecations/changes

- The SparsePanel class has been removed (GH13778)
- The pd.sandbox module has been removed in favor of the external library pandas-qt (GH13670)
- The pandas.io.data and pandas.io.wb modules are removed in favor of the pandas-datareader package (GH13724).
- The pandas.tools.rplot module has been removed in favor of the seaborn package (GH13855)
- DataFrame.to_csv() has dropped the engine parameter, as was deprecated in 0.17.1 (GH11274, GH13419)
- DataFrame.to_dict() has dropped the outtype parameter in favor of orient (GH13627, GH8486)
- pd.Categorical has dropped setting of the ordered attribute directly in favor of the set_ordered method (GH13671)
- pd.Categorical has dropped the levels attribute in favor of categories (GH8376)
- DataFrame.to_sql() has dropped the mysql option for the flavor parameter (GH13611)
- Panel.shift() has dropped the lags parameter in favor of periods (GH14041)
- pd.Index has dropped the diff method in favor of difference (GH13669)
- pd.DataFrame has dropped the to_wide method in favor of to_panel (GH14039)
- Series.to_csv has dropped the nanRep parameter in favor of na_rep (GH13804)
- Series.xs, DataFrame.xs, Panel.xs, Panel.major_xs, and Panel.minor_xs have dropped the copy parameter (GH13781)
- str.split has dropped the return_type parameter in favor of expand (GH13701)
- Removal of the legacy time rules (offset aliases), deprecated since 0.17.0 (this has been alias since 0.8.0) (GH13590, GH13868). Now legacy time rules raises ValueError. For the list of currently supported offsets, see here.
- The default value for the return_type parameter for DataFrame.plot.box and DataFrame.boxplot changed from None to "axes". These methods will now return a matplotlib axes by default instead of a dictionary of artists. See here (GH6581).
- The tquery and uquery functions in the pandas.io.sql module are removed (GH5950).

Performance improvements

- Improved performance of sparse IntIndex.intersect (GH13082)
- Improved performance of sparse arithmetic with BlockIndex when the number of blocks are large, though recommended to use IntIndex in such cases (GH13082)
- Improved performance of DataFrame.quantile() as it now operates per-block (GH11623)
- Improved performance of float64 hash table operations, fixing some very slow indexing and groupby operations in python 3 (GH13166, GH13334)
- Improved performance of DataFrameGroupBy.transform (GH12737)
- Improved performance of Index and Series.duplicated (GH10235)
- Improved performance of Index.difference (GH12044)
- Improved performance of RangeIndex.is_monotonic_increasing and is_monotonic_decreasing (GH13749)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Improved performance of datetime string parsing in `DatetimeIndex` (GH13692)
- Improved performance of hashing `Period` (GH12817)
- Improved performance of factorize of datetime with timezone (GH13750)
- Improved performance of by lazily creating indexing hashtables on larger Indexes (GH14266)
- Improved performance of `groupby.groups` (GH14293)
- Unnecessary materializing of a MultiIndex when introspecting for memory usage (GH14308)

**Bug fixes**

- Bug in `groupby().shift()`, which could cause a segfault or corruption in rare circumstances when grouping by columns with missing values (GH13813)
- Bug in `groupby().cumsum()` calculating `cumprod` when `axis=1` (GH13994)
- Bug in `pd.to_timedelta()` in which the `errors` parameter was not being respected (GH13613)
- Bug in `io.json.json_normalize()`, where non-ascii keys raised an exception (GH13213)
- Bug when passing a not-default-indexed `Series` as `xerr` or `yerr` in `.plot()` (GH11858)
- Bug in area plot draws legend incorrectly if subplot is enabled or legend is moved after plot (matplotlib 1.5.0 is required to draw area plot legend properly) (GH9161, GH13544)
- Bug in `DataFrame` assignment with an object-dtyped `Index` where the resultant column is mutable to the original object. (GH13522)
- Bug in matplotlib `AutoDataFormatter`; this restores the second scaled formatting and re-adds micro-second scaled formatting (GH13131)
- Bug in selection from a `HDFStore` with a fixed format and `start` and/or `stop` specified will now return the selected range (GH8287)
- Bug in `Categorical.from_codes()` where an unhelpful error was raised when an invalid `ordered` parameter was passed in (GH14058)
- Bug in Series construction from a tuple of integers on windows not returning default dtype (int64) (GH13646)
- Bug in `TimedeltaIndex` addition with a Datetime-like object where addition overflow was not being caught (GH14068)
- Bug in `.groupby(..).resample(..)` when the same object is called multiple times (GH13174)
- Bug in `.to_records()` when index name is a unicode string (GH13172)
- Bug in calling `.memory_usage()` on object which doesn’t implement (GH12924)
- Regression in `Series.quantile` with nans (also shows up in `.median()` and `.describe()`); furthermore now names the `Series` with the quantile (GH13098, GH13146)
- Bug in `SeriesGroupBy.transform` with datetime values and missing groups (GH13191)
- Bug where empty `Series` were incorrectly coerced in datetime-like numeric operations (GH13844)
- Bug in `Categorical` constructor when passed a `Categorical` containing datetimes with timezones (GH14190)
- Bug in `Series.str.extractall()` with `str` index raises `ValueError` (GH13156)
- Bug in `Series.str.extractall()` with single group and quantifier (GH13382)
- Bug in `DatetimeIndex` and `Period` subtraction raises `ValueError` or `AttributeError` rather than `TypeError` (GH13078)
• Bug in Index and Series created with NaN and NaT mixed data may not have datetime64 dtype (GH13324)
• Bug in Index and Series may ignore np.datetime64('nat') and np.timedelta64('nat') to infer dtype (GH13324)
• Bug in PeriodIndex and Period subtraction raises AttributeError (GH13071)
• Bug in PeriodIndex construction returning a float64 index in some circumstances (GH13067)
• Bug in .resample(…) with a PeriodIndex not changing its freq appropriately when empty (GH13067)
• Bug in .resample(…) with a PeriodIndex not retaining its type or name with an empty DataFrame appropriately when empty (GH13212)
• Bug in groupby(…).apply(…) when the passed function returns scalar values per group (GH13468).
• Bug in groupby(…).resample(…) where passing some keywords would raise an exception (GH13235)
• Bug in .tz_convert on a tz-aware DateTimeIndex that relied on index being sorted for correct results (GH13066)
• Bug in .tz_localize with dateutil.tz.tzlocal may return incorrect result (GH13583)
• Bug in DatetimeTZDtype dtype with dateutil.tz.tzlocal cannot be regarded as valid dtype (GH13583)
• Bug in pd.read_hdf() where attempting to load an HDF file with a single dataset, that had one or more categorical columns, failed unless the key argument was set to the name of the dataset. (GH13231)
• Bug in rolling() that allowed a negative integer window in construction of the Rolling() object, but would later fail on aggregation (GH13383)
• Bug in Series indexing with tuple-valued data and a numeric index (GH13509)
• Bug in printing pd.DataFrame where unusual elements with the object dtype were causing segfaults (GH13717)
• Bug in ranking Series which could result in segfaults (GH13445)
• Bug in various index types, which did not propagate the name of passed index (GH12309)
• Bug in DatetimeIndex, which did not honour the copy=True (GH13205)
• Bug in DatetimeIndex.is_normalized returns incorrectly for normalized date_range in case of local time-zones (GH13459)
• Bug in pd.concat and .append may coerces datetime64 and timedelta to object dtype containing python built-in datetime or timedelta rather than Timestamp or Timedelta (GH13626)
• Bug in PeriodIndex.append may raises AttributeError when the result is object dtype (GH13221)
• Bug in CategoricalIndex.append may accept normal list (GH13626)
• Bug in pd.concat and .append with the same timezone get reset to UTC (GH7795)
• Bug in Series and DataFrame .append raises AmbiguousTimeError if data contains datetime near DST boundary (GH13626)
• Bug in DataFrame.to_csv() in which float values were being quoted even though quotations were specified for non-numeric values only (GH12922, GH13259)
• Bug in DataFrame.describe() raising ValueError with only boolean columns (GH13898)
• Bug in MultiIndex slicing where extra elements were returned when level is non-unique (GH12896)
• Bug in .str.replace does not raise TypeError for invalid replacement (GH13438)
• Bug in MultiIndex.from_arrays which didn’t check for input array lengths matching (GH13599)
• Bug in cartesian_product and MultiIndex.from_product which may raise with empty input arrays (GH12258)
• Bug in pd.read_csv() which may cause a segfault or corruption when iterating in large chunks over a stream/file under rare circumstances (GH13703)
• Bug in pd.read_csv() which caused errors to be raised when a dictionary containing scalars is passed in for na_values (GH12224)
• Bug in pd.read_csv() which caused BOM files to be incorrectly parsed by not ignoring the BOM (GH4793)
• Bug in pd.read_csv() with engine='python' which raised errors when a numpy array was passed in for usecols (GH12546)
• Bug in pd.read_csv() where the index columns were being incorrectly parsed when parsed as dates with a thousands parameter (GH14066)
• Bug in pd.read_csv() with engine='python' in which NaN values weren't being detected after data was converted to numeric values (GH13314)
• Bug in pd.read_csv() in which the nrows argument was not properly validated for both engines (GH10476)
• Bug in pd.read_csv() with engine='python' in which infinities of mixed-case forms were not being interpreted properly (GH13274)
• Bug in pd.read_csv() with engine='python' in which trailing NaN values were not being parsed (GH13320)
• Bug in pd.read_csv() with engine='python' when reading from a tempfile.TemporaryFile on Windows with Python 3 (GH13398)
• Bug in pd.read_csv() that prevents usecols kwarg from accepting single-byte unicode strings (GH13219)
• Bug in pd.read_csv() that prevents usecols from being an empty set (GH13402)
• Bug in pd.read_csv() in the C engine where the NULL character was not being parsed as NULL (GH14012)
• Bug in pd.read_csv() with engine='c' in which NULL quotechar was not accepted even though quoting was specified as None (GH13411)
• Bug in pd.read_csv() with engine='c' in which fields were not properly cast to float when quoting was specified as non-numeric (GH13411)
• Bug in pd.read_csv() in Python 2.x with non-UTF8 encoded, multi-character separated data (GH3404)
• Bug in pd.read_csv(), where aliases for utf-xx (e.g. UTF-xx, UTF_xx, utf_xx) raised UnicodeDecodeError (GH13549)
• Bug in pd.read_csv, pd.read_table, pd.read_fwf, pd.read_stata and pd.read_sas where files were opened by parsers but not closed if both chunksize and iterator were None. (GH13940)
• Bug in StataReader, StataWriter, XportReader and SAS7BDATReader where a file was not properly closed when an error was raised. (GH13940)
• Bug in pd.pivot_table() where margins_name is ignored when aggfunc is a list (GH13354)
• Bug in pd.Series.str.zfill, center, ljust, rjust, and pad when passing non-integers, did not raise TypeError (GH13598)
• Bug in checking for any null objects in a TimedeltaIndex, which always returned True (GH13603)
• Bug in Series arithmetic raises TypeError if it contains datet ime-like as object dtype (GH13043)
• Bug Series.isnull() and Series.notnull() ignore Period('NaT') (GH13737)
• Bug Series.fillna() and Series.dropna() don't affect to Period('NaT') (GH13737)
• Bug in .fillna(value=np.nan) incorrectly raises KeyError on a category dtype Series (GH14021)
• Bug in extension dtype creation where the created types were not is/identical (GH13285)
• Bug in .resample(..) where incorrect warnings were triggered by IPython introspection (GH13618)
• Bug in NaT - Period raises AttributeError (GH13071)
• Bug in Series comparison may output incorrect result if rhs contains NaT (GH9005)
• Bug in Series and Index comparison may output incorrect result if it contains NaT with object dtype (GH13592)
• Bug in Period addition raises TypeError if Period is on right hand side (GH13069)
• Bug in Period and Series or Index comparison raises TypeError (GH13200)
• Bug in pd.set_eng_float_format() that would prevent NaN and Inf from formatting (GH11981)
• Bug in .unstack with Categorical dtype resets .ordered to True (GH13249)
• Clean some compile time warnings in datetime parsing (GH13607)
• Bug in factorize raises AmbiguousTimeError if data contains datetime near DST boundary (GH13750)
• Bug in .set_index raises AmbiguousTimeError if new index contains DST boundary and multi levels (GH12920)
• Bug in .shift raises AmbiguousTimeError if data contains datetime near DST boundary (GH13926)
• Bug in pd.read_hdf() returns incorrect result when a DataFrame with a categorical column and a query which doesn’t match any values (GH13792)
• Bug in .iloc when indexing with a non lexsorted MultiIndex (GH13797)
• Bug in .loc when indexing with date strings in a reverse sorted DatetimeIndex (GH14316)
• Bug in Series comparison operators when dealing with zero dim NumPy arrays (GH13006)
• Bug in .combine_first may return incorrect dtype (GH7630, GH10567)
• Bug in groupby where apply returns different result depending on whether first result is None or not (GH12824)
• Bug in groupby(...).nth() where the group key is included inconsistently if called after .head()/.tail() (GH12839)
• Bug in .to_html, .to_latex and .to_string silently ignore custom datetime formatter passed through the formatters key word (GH10690)
• Bug in DataFrame.iterrows(), not yielding a Series subclsae if defined (GH13977)
• Bug in pd.to_numeric when errors='coerce' and input contains non-hashable objects (GH13324)
• Bug in invalid Timedelta arithmetic and comparison may raise ValueError rather than TypeError (GH13624)
• Bug in invalid datetime parsing in to_datetime and DatetimeIndex may raise TypeError rather than ValueError (GH11169, GH11287)
• Bug in Index created with tz-aware Timestamp and mismatched tz option incorrectly coerces timezone (GH13692)
• Bug in DatetimeIndex with nanosecond frequency does not include timestamp specified with end (GH13672)
• Bug in `Series when setting a slice with a np.timedelta64 (GH14155)
• Bug in Index raises OutOfBoundsDatetime if datet ime exceeds datetime64[ns] bounds, rather than coercing to object dtype (GH13663)
• Bug in Index may ignore specified datetime64 or timedelta64 passed as dtype (GH13981)
• Bug in RangeIndex can be created without no arguments rather than raises **TypeError** (GH13793)
• Bug in `.value_counts()` raises OutOfBoundsDatetime if data exceeds datetime64[ns] bounds (GH13663)
• Bug in DatetimeIndex may raise OutOfBoundsDatetime if input np.datetime64 has other unit than ns (GH9114)
• Bug in Series creation with np.datetime64 which has other unit than ns as object dtype results in incorrect values (GH13876)
• Bug in resample with timedelta data where data was casted to float (GH13119).
• Bug in pd.isnull() pd.notnull() raise TypeError if input datetime-like has other unit than ns (GH13389)
• Bug in pd.merge() may raise TypeError if input datetime-like has other unit than ns (GH13389)
• Bug in HDFStore/read_hdf() discarded DatetimeIndex.name if tz was set (GH13884)
• Bug in Categorical.remove_unused_categories() changes .codes dtype to platform int (GH13261)
• Bug in groupby with as_index=False returns all NaN’s when grouping on multiple columns including a categorical one (GH13204)
• Bug in df.groupby(...)[...] where getitem with Int64Index raised an error (GH13731)
• Bug in the CSS classes assigned to DataFrame.style for index names. Previously they were assigned "col_heading level<n> col<c>" where n was the number of levels + 1. Now they are assigned "index_name level<n>", where n is the correct level for that MultiIndex.
• Bug where pd.read_gbq() could throw ImportError: No module named discovery as a result of a naming conflict with another python package called apiclient (GH13454)
• Bug in Index.union returns an incorrect result with a named empty index (GH13432)
• Bugs in Index.difference and DataFrame.join raise in Python3 when using mixed-integer indexes (GH13432, GH12814)
• Bug in subtract tz-aware datetime.datetime from tz-aware datetime64 series (GH14088)
• Bug in .to_excel() when DataFrame contains a MultiIndex which contains a label with a NaN value (GH13511)
• Bug in invalid frequency offset string like “D1”, “-2-3H” may not raise ValueError (GH13930)
• Bug in concat and groupby for hierarchical frames with RangeIndex levels (GH13542).
• Bug in Series.str.contains() for Series containing only NaN values of object dtype (GH14171)
• Bug in agg() function on groupby dataframe changes dtype of datetime64[ns] column to float64 (GH12821)
• Bug in using NumPy ufunc with PeriodIndex to add or subtract integer raise IncompatibleFrequency. Note that using standard operator like + or - is recommended, because standard operators use more efficient path (GH13980)
• Bug in operations on NaT returning float instead of datetime64[ns] (GH12941)
• Bug in Series flexible arithmetic methods (like .add()) raises ValueError when axis=None (GH13894)
• Bug in DataFrame.to_csv() with MultiIndex columns in which a stray empty line was added (GH6618)
• Bug in DatetimeIndex, TimedeltaIndex and PeriodIndex.equals() may return True when input isn’t Index but contains the same values (GH13107)
• Bug in assignment against datetime with timezone may not work if it contains datetime near DST boundary (GH14146)
• Bug in pd.eval() and HDFStore query truncating long float literals with python 2 (GH14241)
• Bug in Index raises KeyError displaying incorrect column when column is not in the df and columns contains duplicate values (GH13822)
• Bug in Period and PeriodIndex creating wrong dates when frequency has combined offset aliases (GH13874)
• Bug in .to_string() when called with an integer line_width and index=False raises an UnboundLocalError exception because idx referenced before assignment.
• Bug in eval() where the resolvers argument would not accept a list (GH14095)
• Bugs in stack, get_dummies, make_axis_dummies which don't preserve categorical dtypes in (multi)indexes (GH13854)
• PeriodIndex can now accept list and array which contains pd.NaT (GH13430)
• Bug in df.groupby where .median() returns arbitrary values if grouped dataframe contains empty bins (GH13629)
• Bug in Index.copy() where name parameter was ignored (GH14302)

**Contributors**

A total of 117 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Adrien Emery +
• Alex Alekseyev
• Alex Vig +
• Allen Riddell +
• Amol +
• Amol Agrawal +
• Andy R. Terrel +
• Anthonios Partheniou
• Ben Kandel +
• Bob Baxley +
• Brett Rosen +
• Camilo Cota +
• Chris
• Chris Grinolds
• Chris Warth
• Christian Hudon
• Christopher C. Aycock
• Daniel Siladj +
• Douglas McNeil
• Drewrey Lupton +
• Eduardo Blancas Reyes +
• Elliot Marsden +
• Evan Wright
• Felix Marcinowski +
• Francis T. O’Donovan
• Geraint Duck +
• Giacomo Ferroni +
• Grant Roch +
• Gábor Lipták
• Haleemur Ali +
• Hassan Shamim +
• Iulius Curt +
• Ivan Nazarov +
• Jeff Reback
• Jeffrey Gerard +
• Jenn Olsen +
• Jim Crist
• Joe Jevnik
• John Evans +
• John Freeman
• John Liekezer +
• John W. O’Brien
• John Zwinck +
• Johnny Gill +
• Jordan Erenrich +
• Joris Van den Bossche
• Josh Howes +
• Jozef Brandys +
• Ka Wo Chen
• Kamil Sindi +
• Kerby Shedden
• Kerne +
• Kevin Sheppard
• Matthieu Brucher +
• Maximilian Roos
• Michael Scherer +
• Mike Graham +
• Mortada Mehyar
• Muhammad Haseeb Tariq +
• Nate George +
• Neil Parley +
• Nicolas Bonnotte
• OXPHOS
• Pan Deng / Zora +
• Paul +
• Paul Mestemaker +
• Pauli Virtanen
• Pawel Kordek +
• Pietro Battistoni
• Piotr Jucha +
• Ravi Kumar Nimmi +
• Robert Gieseke
• Robert Kern +
• Roger Thomas
• Roy Keyes +
• Russell Smith +
• Sahil Dua +
• Sanjiv Lobo +
• Sašo Stanovnik +
• Shawn Heide +
• Sinhrks
• Stephen Kappel +
• Steve Choi +
• Stewart Henderson +
• Sudarshan Konge +
• Thomas A Caswell
• Tom Augspurger
• Tom Bird +
• Uwe Hoffmann +
• WillAyd +
• Xiang Zhang +
• YG-Riku +
• Yadunandan +
This is a minor bug-fix release from 0.18.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- `.groupby(...)` has been enhanced to provide convenient syntax when working with `.rolling(...)`, `.expanding(...)` and `.resample(...)` per group, see [here](#).
- `pd.to_datetime()` has gained the ability to assemble dates from a `DataFrame`, see [here](#).
- Method chaining improvements, see [here](#).
- Custom business hour offset, see [here](#).
• Many bug fixes in the handling of sparse, see here
• Expanded the Tutorials section with a feature on modern pandas, courtesy of @TomAugsburger. (GH13045).

What’s new in v0.18.1

• New features
  – Custom business hour
  – Method .groupby(..) syntax with window and resample operations
  – Method chaining improvements
    * Methods .where() and .mask()
    * Methods .loc[], .iloc[], .ix[]
    * Indexing with ‘[]’
  – Partial string indexing on DatetimeIndex when part of a MultiIndex
  – Assembling datetimes
  – Other enhancements
• Sparse changes
• API changes
  – Method .groupby(..).nth() changes
  – NumPy function compatibility
  – Using .apply on GroupBy resampling
  – Changes in read_csv exceptions
  – Method to_datetime error changes
  – Other API changes
  – Deprecations
• Performance improvements
• Bug fixes
• Contributors

New features

Custom business hour

The CustomBusinessHour is a mixture of BusinessHour and CustomBusinessDay which allows you to specify arbitrary holidays. For details, see Custom Business Hour (GH11514)

```python
In [1]: from pandas.tseries.offsets import CustomBusinessHour
In [2]: from pandas.tseries.holiday import USFederalHolidayCalendar
In [3]: bhour_us = CustomBusinessHour(calendar=USFederalHolidayCalendar())
```
Friday before MLK Day

```python
In [4]: import datetime
In [5]: dt = datetime.datetime(2014, 1, 17, 15)
In [6]: dt + bhour_us
Out[6]: Timestamp('2014-01-17 16:00:00')
```

Tuesday after MLK Day (Monday is skipped because it’s a holiday)

```python
In [7]: dt + bhour_us * 2
Out[7]: Timestamp('2014-01-21 09:00:00')
```

**Method `.groupby()` syntax with window and resample operations**

`.groupby(...)` has been enhanced to provide convenient syntax when working with `.rolling(...), .expanding(...) and .resample(...) per group, see (GH12486, GH12738).

You can now use `.rolling(...)` and `.expanding(...)` as methods on groupbys. These return another deferred object (similar to what `.rolling()` and `.expanding()` do on ungrouped pandas objects). You can then operate on these RollingGroupby objects in a similar manner.

Previously you would have to do this to get a rolling window mean per-group:

```python
In [9]: df
Out[9]:
   A B
0  0  0
1  1  1
2  1  2
3  1  3
4  1  4
   ... ..
35 3 35
36 3 36
37 3 37
38 3 38
39 3 39
[40 rows x 2 columns]
```

```python
In [10]: df.groupby("A").apply(lambda x: x.rolling(4).B.mean())
Out[10]:
   A  B
0  0  NaN
1  1  NaN
2  NaN
3  1.5
4  2.5
   ...  
```

(continues on next page)
Now you can do:

```
In [11]: df.groupby("A").rolling(4).B.mean()
Out[11]:
A
1  0    NaN
   1    NaN
   2    NaN
   3  1.5
   4  2.5
   ...
3 35  33.5
36 34.5
37 35.5
38 36.5
39 37.5
Name: B, Length: 40, dtype: float64
```

For `.resample(...)` type of operations, previously you would have to:

```
In [12]: df = pd.DataFrame(
           {
              "date": pd.date_range(start="2016-01-01", periods=4, freq="W"),
              "group": [1, 1, 2, 2],
              "val": [5, 6, 7, 8],
           }).set_index("date")
In [13]: df
Out[13]:
    group  val
date
2016-01-03  1  5
2016-01-10  1  6
2016-01-17  2  7
2016-01-24  2  8
[4 rows x 2 columns]
```

```
In [14]: df.groupby("group").apply(lambda x: x.resample("1D").ffill())
Out[14]:
    group  date  val
       1  2016-01-03  1  5
```

(continues on next page)
Now you can do:

```python
In [15]: df.groupby("group").resample("1D").ffill()
Out[15]:
              group  val
        date
1 2016-01-03   1   5
   2016-01-04   1   5
   2016-01-05   1   5
   2016-01-06   1   5
   2016-01-07   1   5
   ...       ...  ...
2 2016-01-20   2   7
   2016-01-21   2   7
   2016-01-22   2   7
   2016-01-23   2   7
   2016-01-24   2   8
[16 rows x 2 columns]
```

**Method chaining improvements**

The following methods / indexers now accept a callable. It is intended to make these more useful in method chains, see the documentation. (GH11485, GH12533)

- `.where()` and `.mask()`
- `.loc[]`, `.iloc[]` and `.ix[]`
- [] indexing
Methods `.where()` and `.mask()`

These can accept a callable for the condition and other arguments.

```python
In [16]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})

In [17]: df.where(lambda x: x > 4, lambda x: x + 10)
Out[17]:
   A  B  C
0  11 14  7
1  12  5  8
2  13  6  9
[3 rows x 3 columns]
```

Methods `.loc[]`, `.iloc[]`, `.ix[]`

These can accept a callable, and a tuple of callable as a slicer. The callable can return a valid boolean indexer or anything which is valid for these indexer’s input.

```python
# callable returns bool indexer
In [18]: df.loc[lambda x: x.A >= 2, lambda x: x.sum() > 10]
Out[18]:
   B  C
1  5  8
2  6  9
[2 rows x 2 columns]

# callable returns list of labels
In [19]: df.loc[lambda x: [1, 2], lambda x: ['A', 'B']]
Out[19]:
   A  B
1  2  5
2  3  6
[2 rows x 2 columns]
```

Indexing with `[]`

Finally, you can use a callable in `[]` indexing of Series, DataFrame and Panel. The callable must return a valid input for `[]` indexing depending on its class and index type.

```python
In [20]: df[lambda x: "A"]
Out[20]:
   0  1
  1  2
  2  3
Name: A, Length: 3, dtype: int64
```

Using these methods / indexers, you can chain data selection operations without using temporary variable.
In [21]: `bb = pd.read_csv("data/baseball.csv", index_col="id")`

In [22]: `(bb.groupby(["year", "team"]).sum().loc[lambda df: df.r > 100])`

Out[22]:

<table>
<thead>
<tr>
<th></th>
<th>stint</th>
<th>g</th>
<th>ab</th>
<th>r</th>
<th>h</th>
<th>X2b</th>
<th>X3b</th>
<th>hr</th>
<th>rbi</th>
<th>sb</th>
<th>cs</th>
<th>bb</th>
<th>so</th>
<th>ibb</th>
<th>hbp</th>
<th>sh</th>
<th>sf</th>
<th>gidp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>team</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>CIN</td>
<td>6</td>
<td>379</td>
<td>745</td>
<td>101</td>
<td>203</td>
<td>35</td>
<td>2</td>
<td>36</td>
<td>125.0</td>
<td>1.0</td>
<td>1.0</td>
<td>105</td>
<td>127.0</td>
<td>14.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0</td>
<td>15.0</td>
<td>18.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10.0</td>
<td>4.0</td>
<td>8.0</td>
<td>28.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.0</td>
<td>16.0</td>
<td>6.0</td>
<td>17.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.0</td>
<td>3.0</td>
<td>8.0</td>
<td>29.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.0</td>
<td>18.0</td>
<td>15.0</td>
<td>48.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>482</td>
<td>1305</td>
<td>198</td>
<td>337</td>
<td>67</td>
<td>6</td>
<td>40</td>
<td>171.0</td>
<td>26.0</td>
<td>7.0</td>
<td>235</td>
<td>188.0</td>
<td>51.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>105</td>
<td>185</td>
<td>115</td>
<td>200</td>
<td>40</td>
<td>4</td>
<td>28</td>
<td>115.0</td>
<td>21.0</td>
<td>4.0</td>
<td>73</td>
<td>140.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
<td>20</td>
<td>8.0</td>
<td>16.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12.0</td>
<td>4.0</td>
<td>16.0</td>
<td>38.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[8 rows x 18 columns]

Partial string indexing on DatetimeIndex when part of a MultiIndex

Partial string indexing now matches on DateTimeIndex when part of a MultiIndex (GH10331)

In [23]: `dft2 = pd.DataFrame(
        ....:     np.random.randn(20, 1),
        ....:     columns=["A"],
        ....:     index=pd.MultiIndex.from_product(
        ....:         [pd.date_range("20130101", periods=10, freq="12H"), ["a", "b"]]
        ....:     ),
        ....: )
        ....: `n

In [24]: `dft2`

Out[24]:

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 00:00:00 a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>2013-01-01 12:00:00 a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>2013-01-02 00:00:00 a</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>2013-01-04 12:00:00 b</td>
</tr>
<tr>
<td>2013-01-05 00:00:00 a</td>
</tr>
</tbody>
</table>

(continues on next page)
In [25]: dft2.loc["2013-01-05"]
Out[25]:
A  
2013-01-05 00:00:00 a -0.424972
     b  0.567020
2013-01-05 12:00:00 a  0.276232
     b -1.087401
[4 rows x 1 columns]

On other levels

In [26]: idx = pd.IndexSlice

In [27]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [28]: dft2
Out[28]:
A  
a 2013-01-01 00:00:00 0.469112
   2013-01-01 12:00:00 -1.509059
   2013-01-02 00:00:00 1.212112
   2013-01-02 12:00:00 0.119209
   2013-01-03 00:00:00 -0.861849
   ... ... ... ... ...
   b 2013-01-03 12:00:00 1.071804
   2013-01-04 00:00:00 -0.706771
   2013-01-04 12:00:00 0.271860
   2013-01-05 00:00:00 0.567020
   2013-01-05 12:00:00 -1.087401
[20 rows x 1 columns]

In [29]: dft2.loc[idx[:, "2013-01-05"], :]
Out[29]:
A  
a 2013-01-05 00:00:00 -0.424972
   2013-01-05 12:00:00 0.276232
   b 2013-01-05 00:00:00 0.567020
   2013-01-05 12:00:00 -1.087401
[4 rows x 1 columns]
Assembling datetimes

`pd.to_datetime()` has gained the ability to assemble datetimes from a passed in `DataFrame` or a dict. (GH8158).

```
In [30]: df = pd.DataFrame(
    ....:   {
    ....:     "year": [2015, 2016],
    ....:     "month": [2, 3],
    ....:     "day": [4, 5],
    ....:     "hour": [2, 3]
    ....:   }
    ....:
    ....:)
```

```
In [31]: df
Out[31]:
year  month  day  hour
0  2015     2    4    2
1  2016     3    5    3
[2 rows x 4 columns]
```

Assembling using the passed frame.

```
In [32]: pd.to_datetime(df)
Out[32]:
0  2015-02-04 02:00:00
1  2016-03-05 03:00:00
Length: 2, dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```
In [33]: pd.to_datetime(df[['year', 'month', 'day']])
Out[33]:
0  2015-02-04
1  2016-03-05
Length: 2, dtype: datetime64[ns]
```

Other enhancements

- `pd.read_csv()` now supports `delim_whitespace=True` for the Python engine (GH12958)
- `pd.read_csv()` now supports opening ZIP files that contains a single CSV, via extension inference or explicit `compression='zip'` (GH12175)
- `pd.read_csv()` now supports opening files using xz compression, via extension inference or explicit `compression='xz'` is specified; xz compressions is also supported by `DataFrame.to_csv` in the same way (GH11852)
- `pd.read_msgpack()` now always gives writeable ndarrays even when compression is used (GH12359).
- `pd.read_msgpack()` now supports serializing and de-serializing categoricals with msgpack (GH12573)
- `.to_json()` now supports `NDFrames` that contain categorical and sparse data (GH10778)
- `interpolate()` now supports method='akima' (GH7588).
- `pd.read_excel()` now accepts path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path, in line with other `read_*` functions (GH12655)
- Added `.weekday_name` property as a component to `DatetimeIndex` and the `.dt` accessor. (GH11128)
• Index.take now handles allow_fill and fill_value consistently (GH12631)

```python
In [34]: idx = pd.Index([1.0, 2.0, 3.0, 4.0], dtype="float")
# default, allow_fill=True, fill_value=None
In [35]: idx.take([2, -1])
Out[35]: Float64Index([3.0, 4.0], dtype='float64')
In [36]: idx.take([2, -1], fill_value=True)
Out[36]: Float64Index([3.0, nan], dtype='float64')
```

• Index now supports .str.get_dummies() which returns MultiIndex, see Creating Indicator Variables (GH10008, GH10103)

```python
In [37]: idx = pd.Index(["a|b", "a|c", "b|c"])
In [38]: idx.str.get_dummies("|")
Out[38]: MultiIndex([(1, 1, 0),
                   (1, 0, 1),
                   (0, 1, 1)],
                   names=["a", "b", "c"])
```

• pd.crosstab() has gained a normalize argument for normalizing frequency tables (GH12569). Examples in the updated docs here.

• .resample(..).interpolate() is now supported (GH12925)

• .isin() now accepts passed sets (GH12988)

Sparse changes

These changes conform sparse handling to return the correct types and work to make a smoother experience with indexing.

SparseArray.take now returns a scalar for scalar input, SparseArray for others. Furthermore, it handles a negative indexer with the same rule as Index (GH10560, GH12796)

```python
s = pd.SparseArray([np.nan, np.nan, 1, 2, 3, np.nan, 4, 5, np.nan, 6])
s.take(0)
s.take([1, 2, 3])
```

• Bug in SparseSeries[] indexing with Ellipsis raises KeyError (GH9467)

• Bug in SparseArray[] indexing with tuples are not handled properly (GH12966)

• Bug in SparseSeries.loc[] with list-like input raises TypeError (GH10560)

• Bug in SparseSeries.iloc[] with scalar input may raise IndexError (GH10560)

• Bug in SparseSeries.loc[].iloc[] with slice returns SparseArray, rather than SparseSeries (GH10560)

• Bug in SparseDataFrame.loc[], .iloc[] may results in dense Series, rather than SparseSeries (GH12787)

• Bug in SparseArray addition ignores fill_value of right hand side (GH12910)

• Bug in SparseArray mod raises AttributeError (GH12910)
• Bug in SparseArray pow calculates $1 ** \text{np.nan}$ as np.nan which must be 1 (GH12910)
• Bug in SparseArray comparison output may incorrect result or raise ValueError (GH12971)
• Bug in SparseSeries.__repr__ raises TypeError when it is longer than max_rows (GH10560)
• Bug in SparseSeries.shape ignores fill_value (GH10452)
• Bug in SparseSeries and SparseArray may have different dtype from its dense values (GH12908)
• Bug in SparseSeries.to_frame() results in DataFrame, rather than SparseDataFrame (GH9850)
• Bug in SparseSeries.value_counts() does not count fill_value (GH6749)
• Bug in SparseArray.to_dense() does not preserve dtype (GH10648)
• Bug in SparseSeries.to_dense() incorrectly handle fill_value (GH12797)
• Bug in pd.concat() of SparseSeries results in dense (GH10536)
• Bug in pd.concat() of SparseDataFrame incorrectly handle fill_value (GH9765)
• Bug in pd.concat() of SparseDataFrame may raise AttributeError (GH12174)
• Bug in SparseArray.shift() may raise NameError or TypeError (GH12908)

API changes

Method .groupby(..).nth() changes

The index in .groupby(..).nth() output is now more consistent when the as_index argument is passed (GH11039):

```python
In [39]: df = pd.DataFrame({"A": ["a", "b", "a"], "B": [1, 2, 3]})
In [40]: df
Out[40]:
     A  B
0    a  1
1    b  2
2    a  3
[3 rows x 2 columns]
```

Previous behavior:

```python
In [3]: df.groupby('A', as_index=True)['B'].nth(0)
Out[3]:
 0  1
1  2
Name: B, dtype: int64

In [4]: df.groupby('A', as_index=False)['B'].nth(0)
Out[4]:
 0  1
1  2
Name: B, dtype: int64
```
New behavior:

```
In [41]: df.groupby("A", as_index=True)["B"][0]
Out[41]:
   A
  a  1
  b  2
Name: B, Length: 2, dtype: int64

In [42]: df.groupby("A", as_index=False)["B"][0]
Out[42]:
   0  1
   1  2
Name: B, Length: 2, dtype: int64
```

Furthermore, previously, a .groupby would always sort, regardless if sort=False was passed with .nth().

```
In [43]: np.random.seed(1234)
In [44]: df = pd.DataFrame(np.random.randn(100, 2), columns=["a", "b"])
In [45]: df["c"] = np.random.randint(0, 4, 100)
Previous behavior:
In [4]: df.groupby( "c", sort=True).nth(1)
Out[4]:
   a    b
  c
  0  -0.334077  0.002118
  1   0.036142 -2.074978
  2  -0.720589  0.887163
  3   0.859588 -0.636524

In [5]: df.groupby( "c", sort=False).nth(1)
Out[5]:
   a    b
  c
  0  -0.334077  0.002118
  1   0.036142 -2.074978
  2  -0.720589  0.887163
  3   0.859588 -0.636524
```

New behavior:

```
In [46]: df.groupby("c", sort=True).nth(1)
Out[46]:
   a    b
  c
  0  -0.334077  0.002118
  1   0.036142 -2.074978
  2  -0.720589  0.887163
  3   0.859588 -0.636524
```

(continues on next page)
In [47]: df.groupby("c", sort=False).nth(1)
Out[47]:
   a    b
   c
   2 -0.720589  0.887163
   3  0.859588 -0.636524
   0 -0.334077  0.002118
   1  0.036142 -2.074978

[4 rows x 2 columns]

NumPy function compatibility

Compatibility between pandas array-like methods (e.g. sum and take) and their numpy counterparts has been greatly increased by augmenting the signatures of the pandas methods so as to accept arguments that can be passed in from numpy, even if they are not necessarily used in the pandas implementation (GH12644, GH12638, GH12687)

• .searchsorted() for Index and TimedeltaIndex now accept a sorter argument to maintain compatibility with numpy’s searchsorted function (GH12238)

• Bug in numpy compatibility of np.round() on a Series (GH12600)

An example of this signature augmentation is illustrated below:

sp = pd.SparseDataFrame([1, 2, 3])
sp

Previous behaviour:

In [2]: np.cumsum(sp, axis=0)
...:
TypeError: cumsum() takes at most 2 arguments (4 given)

New behaviour:

np.cumsum(sp, axis=0)

Using .apply on GroupBy resampling

Using apply on resampling groupby operations (using a pd.TimeGrouper) now has the same output types as similar apply calls on other groupby operations. (GH11742).

In [48]: df = pd.DataFrame(
       ...:     {"date": pd.to_datetime(["10/10/2000", "11/10/2000"]), "value": [10, 13]}
       ...: )
       ...:

In [49]: df
Out[49]:

df

<table>
<thead>
<tr>
<th>date</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-10-10</td>
<td>10</td>
</tr>
<tr>
<td>2000-11-10</td>
<td>13</td>
</tr>
</tbody>
</table>

[2 rows x 2 columns]

Previous behavior:

```python
In [1]: df.groupby(pd.TimeGrouper(key='date',
                            ...: freq='M')).apply(lambda x: x.value.sum())
Out[1]:
...
TypeError: cannot concatenate a non-NDFrame object
```

# Output is a Series
```
In [2]: df.groupby(pd.TimeGrouper(key='date',
                                ...: freq='M')).apply(lambda x: x[['value']].sum())
Out[2]:
date         value
2000-10-31    10
2000-11-30    13
```

dtype: int64

New behavior:

# Output is a Series
```
In [55]: df.groupby(pd.TimeGrouper(key='date',
                                 ...: freq='M')).apply(lambda x: x.value.sum())
Out[55]:
date
2000-10-31    10
2000-11-30    13
Freq: M, dtype: int64
```

# Output is a DataFrame
```
In [56]: df.groupby(pd.TimeGrouper(key='date',
                                  ...: freq='M')).apply(lambda x: x[['value']].sum())
Out[56]:
            value
date
2000-10-31    10
2000-11-30    13
```
Changes in read_csv exceptions

In order to standardize the read_csv API for both the c and python engines, both will now raise an EmptyDataError, a subclass of ValueError, in response to empty columns or header (GH12493, GH12506)

Previous behaviour:

```python
In [1]: import io
In [2]: df = pd.read_csv(io.StringIO(''), engine='c')
...
ValueError: No columns to parse from file
In [3]: df = pd.read_csv(io.StringIO(''), engine='python')
...
StopIteration
```

New behaviour:

```python
In [1]: df = pd.read_csv(io.StringIO(''), engine='c')
...
pandas.io.common.EmptyDataError: No columns to parse from file
In [2]: df = pd.read_csv(io.StringIO(''), engine='python')
...
pandas.io.common.EmptyDataError: No columns to parse from file
```

In addition to this error change, several others have been made as well:

- CParserError now sub-classes ValueError instead of just a Exception (GH12551)
- A CParserError is now raised instead of a generic Exception in read_csv when the c engine cannot parse a column (GH12506)
- A ValueError is now raised instead of a generic Exception in read_csv when the c engine encounters a NaN value in an integer column (GH12506)
- A ValueError is now raised instead of a generic Exception in read_csv when true_values is specified, and the c engine encounters an element in a column containing unencodable bytes (GH12506)
- pandas.parser.OverflowError exception has been removed and has been replaced with Python’s built-in OverflowError exception (GH12506)
- pd.read_csv() no longer allows a combination of strings and integers for the usecols parameter (GH12678)

Method to_datetime error changes

Bugs in pd.to_datetime() when passing a unit with convertible entries and errors='coerce' or non-convertible with errors='ignore'. Furthermore, an OutOfBoundsDateime exception will be raised when an out-of-range value is encountered for that unit when errors='raise'. (GH11758, GH13052, GH13059)

Previous behaviour:

```python
In [27]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[27]: NaT
```
In [28]: pd.to_datetime(11111111, unit='D', errors='ignore')
OverflowError: Python int too large to convert to C long

In [29]: pd.to_datetime(11111111, unit='D', errors='raise')
OverflowError: Python int too large to convert to C long

New behaviour:

In [2]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[2]: Timestamp('2014-12-31 16:31:00')

In [3]: pd.to_datetime(11111111, unit='D', errors='ignore')
Out[3]: 11111111

In [4]: pd.to_datetime(11111111, unit='D', errors='raise')
OutOfBoundsDatetime: cannot convert input with unit 'D'

Other API changes

- .swaplevel() for Series, DataFrame, Panel, and MultiIndex now features defaults for its first two parameters i and j that swap the two innermost levels of the index. (GH12934)
- .searchsorted() for Index and TimedeltaIndex now accept a sorter argument to maintain compatibility with numpy's searchsorted function (GH12238)
- Period and PeriodIndex now raises IncompatibleFrequency error which inherits ValueError rather than raw ValueError (GH12615)
- Series.apply for category dtype now applies the passed function to each of the .categories (and not the .codes), and returns a category dtype if possible (GH12473)
- read_csv will now raise a TypeError if parse_dates is neither a boolean, list, or dictionary (matches the doc-string) (GH5636)
- The default for .query()/.eval() is now engine=None, which will use numexpr if it's installed; otherwise it will fallback to the python engine. This mimics the pre-0.18.1 behavior if numexpr is installed (and which, previously, if numexpr was not installed, .query()/.eval() would raise). (GH12749)
- pd.show_versions() now includes pandas_datareader version (GH12740)
- Provide a proper __name__ and __qualname__ attributes for generic functions (GH12021)
- pd.concat(ignore_index=True) now uses RangeIndex as default (GH12695)
- pd.merge() and DataFrame.join() will show a UserWarning when merging/joining a single- with a multi-leveled dataframe (GH9455, GH12219)
- Compat with scipy > 0.17 for deprecated piecewise_polynomial interpolation method; support for the replacement from_derivatives method (GH12887)
Deprecations

- The method name `Index.sym_diff()` is deprecated and can be replaced by `Index.symmetric_difference()` (GH12591)
- The method name `Categorical.sort()` is deprecated in favor of `Categorical.sort_values()` (GH12882)

Performance improvements

- Improved speed of SAS reader (GH12656, GH12961)
- Performance improvements in `.groupby(..).cumcount()` (GH11039)
- Improved memory usage in `pd.read_csv()` when using `skiprows=an_integer` (GH13005)
- Improved performance of `DataFrame.to_sql` when checking case sensitivity for tables. Now only checks if table has been created correctly when table name is not lower case. (GH12876)
- Improved performance of `Period` construction and time series plotting (GH12903, GH11831).
- Improved performance of `.str.encode()` and `.str.decode()` methods (GH13008)
- Improved performance of `to_numeric` if input is numeric dtype (GH12777)
- Improved performance of sparse arithmetic with `IntIndex` (GH13036)

Bug fixes

- `usecols` parameter in `pd.read_csv` is now respected even when the lines of a CSV file are not even (GH12203)
- Bug in `groupby.transform(..)` when `axis=1` is specified with a non-monotonic ordered index (GH12713)
- Bug in `Period` and `PeriodIndex` creation raises `KeyError` if `freq="Minute"` is specified. Note that “Minute” freq is deprecated in v0.17.0, and recommended to use `freq="T"` instead (GH11854)
- Bug in `.resample(...).count()` with a `PeriodIndex` always raising a `TypeError` (GH12774)
- Bug in `.resample(...)` with a `PeriodIndex` casting to a `DatetimeIndex` when empty (GH12868)
- Bug in `.resample(...)` when resampling to an existing frequency (GH12770)
- Bug in printing data which contains `Period` with different `freq` raises `ValueError` (GH12615)
- Bug in `Series` construction with `Categorical` and `dtype='category'` is specified (GH12574)
- Bugs in concatenation with a coercible dtype was too aggressive, resulting in different dtypes in output formatting when an object was longer than `display.max_rows` (GH12411, GH12045, GH11594, GH10571, GH12211)
- Bug in `float_format` option with option not being validated as a callable. (GH12706)
- Bug in `GroupBy.filter` when `dropna=False` and no groups fulfilled the criteria (GH12768)
- Bug in `__name__` of `.cum*` functions (GH12021)
- Bug in `.astype()` of a `Float64Inde/Int64Index` to an `Int64Index` (GH12881)
- Bug in round tripping an integer based index in `.to_json()/.read_json()` when `orient='index'` (the default) (GH12866)
- Bug in plotting `Categorical` dtypes cause error when attempting stacked bar plot (GH13019)
- Compat with >= numpy 1.11 for `NaT` comparisons (GH12969)
- Bug in `.drop()` with a non-unique `MultiIndex`. (GH12701)
• Bug in `concat` of datetime tz-aware and naive DataFrames (GH12467)
• Bug in correctly raising a `ValueError` in `resample(...)`.fillna(...) when passing a non-string (GH12952)
• Bug fixes in various encoding and header processing issues in `pd.read_sas()` (GH12659, GH12654, GH12647, GH12809)
• Bug in `pd.crosstab()` where would silently ignore `aggfunc` if `values=None` (GH12569).
• Potential segfault in `DataFrame.to_json` when serialising `datetime.time` (GH11473).
• Potential segfault in `DataFrame.to_json` when attempting to serialise 0d array (GH11299).
• Segfault in `to_json` when attempting to serialise a `DataFrame` or `Series` with non-ndarray values; now supports serialization of category, sparse, and `datetime64[ns, tz]` dtypes (GH10778).
• Bug in `DataFrame.to_json` with unsupported dtype not passed to default handler (GH12554).
• Bug in `-align` not returning the sub-class (GH12983)
• Bug in aligning a Series with a `DataFrame` (GH13037)
• Bug in ABCPanel in which Panel4D was not being considered as a valid instance of this generic type (GH12810)
• Bug in consistency of `.name` on `.groupby(...)`.apply(...) cases (GH12363)
• Bug in `Timestamp.__repr__` that caused `pprint` to fail in nested structures (GH12622)
• Bug in Timedelta.min and Timedelta.max, the properties now report the true minimum/maximum timedeltas as recognized by pandas. See the documentation. (GH12727)
• Bug in `.quantile()` with interpolation may coerce to float unexpectedly (GH12772)
• Bug in `.quantile()` with empty Series may return scalar rather than empty Series (GH12772)
• Bug in `.loc` with out-of-bounds in a large indexer would raise `IndexError` rather than `KeyError` (GH12527)
• Bug in resampling when using a `TimedeltaIndex` and `.asfreq()`, would previously not include the final fencepost (GH12926)
• Bug in equality testing with a `Categorical` in a `DataFrame` (GH12564)
• Bug in GroupBy.first(), .last() returns incorrect row when `TimeGrouper` is used (GH7453)
• Bug in `pd.read_csv()` with the c engine when specifying `skiprows` with newlines in quoted items (GH10911, GH12775)
• Bug in `DataFrame` timezone lost when assigning tz-aware `datetime` `Series` with alignment (GH12981)
• Bug in `.value_counts()` when normalize=True and dropna=True where nulls still contributed to the normalized count (GH12558)
• Bug in `Series.value_counts()` loses name if its dtype is category (GH12835)
• Bug in `Series.value_counts()` loses timezone info (GH12835)
• Bug in `Series.value_counts(normalize=True)` with `Categorical` raises `UnboundLocalError` (GH12835)
• Bug in `Panel.fillna()` ignoring `inplace=True` (GH12633)
• Bug in `pd.read_csv()` when specifying names, `usecols`, and `parse_dates` simultaneously with the c engine (GH9755)
• Bug in `pd.read_csv()` when specifying `delim_whitespace=True` and `lineterminator` simultaneously with the c engine (GH12912)
- Bug in Series.rename, DataFrame.rename and DataFrame.rename_axis not treating Series as mappings to relabel (GH12623).
- Clean in .rolling.min and .rolling.max to enhance dtype handling (GH12373)
- Bug in groupby where complex types are coerced to float (GH12902)
- Bug in Series.map raises TypeError if its dtype is category or tz-aware datetime (GH12473)
- Bugs on 32bit platforms for some test comparisons (GH12972)
- Bug in index coercion when falling back from RangeIndex construction (GH12893)
- Better error message in window functions when invalid argument (e.g. a float window) is passed (GH12669)
- Bug in slicing subclassed DataFrame defined to return subclassed Series may return normal Series (GH11559)
- Bug in .str accessor methods may raise ValueError if input has name and the result is DataFrame or MultiIndex (GH12617)
- Bug in DataFrame.last_valid_index() and DataFrame.first_valid_index() on empty frames (GH12800)
- Bug in CategoricalIndex.get_loc returns different result from regular Index (GH12531)
- Bug in PeriodIndex.resample where name not propagated (GH12769)
- Bug in date_range closed keyword and timezones (GH12684).
- Bug in pd.concat raises AttributeError when input data contains tz-aware datetime and timedelta (GH12620)
- Bug in pd.concat did not handle empty Series properly (GH11082)
- Bug in .plot.bar alignment when width is specified with int (GH12979)
- Bug in fill_value is ignored if the argument to a binary operator is a constant (GH12723)
- Bug in pd.read_html() when using bs4 flavor and parsing table with a header and only one column (GH9178)
- Bug in .pivot_table when margins=True and dropna=True where nulls still contributed to margin count (GH12577)
- Bug in .pivot_table when dropna=False where table index/column names disappear (GH12133)
- Bug in pd.crosstab() when margins=True and dropna=False which raised (GH12642)
- Bug in Series.name when name attribute can be a hashable type (GH12610)
- Bug in .describe() resets categorical columns information (GH11558)
- Bug where loffset argument was not applied when calling resample().count() on a timeseries (GH12725)
- pd.read_excel() now accepts column names associated with keyword argument names (GH12870)
- Bug in pd.to_numeric() with Index returns np.ndarray, rather than Index (GH12777)
- Bug in pd.to_numeric() with datetime-like may raise TypeError (GH12777)
- Bug in pd.to_numeric() with scalar raises ValueError (GH12777)
Contributors

A total of 60 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Andrew Fiore-Gartland +
- Bastiaan +
- Benoît Vinot +
- Brandon Rhodes +
- DaCoEx +
- Drew Fustin +
- Ernesto Freitas +
- Filip Ter +
- Gregory Livschitz +
- Gábor Lipták
- Hassan Kibirige +
- Iblis Lin
- Israel Saeta Pérez +
- Jason Wolosonovich +
- Jeff Reback
- Joe Jevnik
- Joris Van den Bossche
- Joshua Storck +
- Ka Wo Chen
- Kerby Shedden
- Kieran O’Mahony
- Leif Walsh +
- Mahmoud Lababidi +
- Maoyuan Liu +
- Mark Roth +
- Matt Wittmann
- MaxU +
- Maximilian Roos
- Michael Droettboom +
- Nick Eubank
- Nicolas Bonnotte
- OXPHOS +
- Pauli Virtanen +
5.13.2 Version 0.18.0 (March 13, 2016)

This is a major release from 0.17.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Warning: pandas >= 0.18.0 no longer supports compatibility with Python version 2.6 and 3.3 (GH7718, GH11273)
**Warning:** `numexpr` version 2.4.4 will now show a warning and not be used as a computation back-end for pandas because of some buggy behavior. This does not affect other versions (>= 2.1 and >= 2.4.6). (GH12489)

Highlights include:

- Moving and expanding window functions are now methods on Series and DataFrame, similar to `.groupby`, see [here](#).
- Adding support for a `RangeIndex` as a specialized form of the `Int64Index` for memory savings, see [here](#).
- API breaking change to the `.resample` method to make it more `.groupby` like, see [here](#).
- Removal of support for positional indexing with floats, which was deprecated since 0.14.0. This will now raise a `TypeError`, see [here](#).
- The `.to_xarray()` function has been added for compatibility with the `xarray` package, see [here](#).
- The `read_sas` function has been enhanced to read `sas7bdat` files, see [here](#).
- Addition of the `.str.extractall()` method, and API changes to the `.str.extract()` method and `.str.cat()` method.
- `pd.test()` top-level nose test runner is available (GH4327).

Check the [API Changes and deprecations](#) before updating.

---

**What’s new in v0.18.0**

- **New features**
  - Window functions are now methods
  - Changes to rename
  - Range Index
  - Changes to str.extract
  - Addition of str.extractall
  - Changes to str.cat
  - Datetimelike rounding
  - Formatting of integers in FloatIndex
  - Changes to dtype assignment behaviors
  - Method to _xarray
  - Latex representation
  - `pd.read_sas()` changes
  - Other enhancements

- **Backwards incompatible API changes**
  - NaT and Timedelta operations
  - Changes to msgpack
  - Signature change for `.rank`
  - Bug in QuarterBegin with n=0
Resample API

* Downsampling
* Upsampling
* Previous API will work but with deprecations

Changes to eval

Other API changes

Deprecations

Removal of deprecated float indexers

Removal of prior version deprecations/changes

• Performance improvements
• Bug fixes
• Contributors

New features

Window functions are now methods

Window functions have been refactored to be methods on Series/DataFrame objects, rather than top-level functions, which are now deprecated. This allows these window-type functions, to have a similar API to that of .groupby. See the full documentation here (GH11603, GH12373)

In [1]: np.random.seed(1234)

In [2]: df = pd.DataFrame({'A': range(10), 'B': np.random.randn(10)})

In [3]: df

Out[3]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.471435</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1.190976</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.432707</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-0.312652</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-0.720589</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.887163</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0.859588</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>-0.636524</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0.015696</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>-2.242685</td>
</tr>
</tbody>
</table>

[10 rows x 2 columns]

Previous behavior:

In [8]: pd.rolling_mean(df, window=3)

FutureWarning: pd.rolling_mean is deprecated for DataFrame and will be removed in a future version, replace with
**DataFrame.rolling(window=3,center=False).mean()**

```
Out[8]:
          A     B
        0     NaN  NaN
        1     NaN  NaN
        2   1.0  0.237722
        3   2.0 -0.023640
        4   3.0  0.133155
        5   4.0 -0.048693
        6   5.0  0.342054
        7   6.0  0.370076
        8   7.0  0.079587
        9   8.0 -0.954504
```

New behavior:

```
In [4]: r = df.rolling(window=3)
```

These show a descriptive repr

```
In [5]: r
Out[5]: Rolling [window=3,center=False,axis=0,method=single]
```

with tab-completion of available methods and properties.

```
In [9]: r.<TAB>  # noqa E225, E999
r.A             r.name  r.agg  r.max  r.count  r.min  r.agg  r.name  r.min  r.max  r.count  r.aggregate  r.mean  r.median  
               r.name  r.agg  r.max  r.count  r.min  r.agg  r.name  r.min  r.max  r.count  r.aggregate  r.mean  r.median  
```

The methods operate on the `Rolling` object itself

```
In [6]: r.mean()
Out[6]:

          A     B
        0     NaN  NaN
        1     NaN  NaN
        2  1.0  0.237722
        3  2.0 -0.023640
        4  3.0  0.133155
        5  4.0 -0.048693
        6  5.0  0.342054
        7  6.0  0.370076
        8  7.0  0.079587
        9  8.0 -0.954504
[10 rows x 2 columns]
```

They provide getitem accessors

```
In [7]: r['A'].mean()
Out[7]:
```

(continues on next page)
And multiple aggregations

```python
In [8]: r.agg({'A': ['mean', 'std'], 'B': ['mean', 'std']})
```

```text
Out[8]:

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>mean</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>0.2377</td>
<td>1.33</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>1.0</td>
<td>-0.02</td>
<td>1.34</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>1.0</td>
<td>0.13</td>
<td>1.14</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>1.0</td>
<td>-0.05</td>
<td>0.84</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>1.0</td>
<td>0.34</td>
<td>0.92</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>1.0</td>
<td>0.37</td>
<td>0.87</td>
</tr>
<tr>
<td>7</td>
<td>7.0</td>
<td>1.0</td>
<td>0.08</td>
<td>0.75</td>
</tr>
<tr>
<td>8</td>
<td>8.0</td>
<td>1.0</td>
<td>-0.95</td>
<td>1.16</td>
</tr>
</tbody>
</table>
```

[10 rows x 4 columns]

**Changes to rename**

`Series.rename` and `NDFrame.rename_axis` can now take a scalar or list-like argument for altering the Series or axis name, in addition to their old behaviors of altering labels. (GH9494, GH11965)

```python
In [9]: s = pd.Series(np.random.randn(5))

In [10]: s.rename('newname')
```

```text
Out[10]:

<table>
<thead>
<tr>
<th></th>
<th>newname</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.15</td>
</tr>
<tr>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>-2.02</td>
</tr>
<tr>
<td>4</td>
<td>-0.33</td>
</tr>
</tbody>
</table>
```

Name: newname, Length: 5, dtype: float64

```python
In [11]: df = pd.DataFrame(np.random.randn(5, 2))
```
In [12]: (df.rename_axis("indexname")
.....: .rename_axis("columns_name", axis="columns"))
.....:

Out[12]:
columns_name     0   1
indexname
0   0.002118  0.405453
1   0.289092  1.321158
2  -1.546906 -0.202646
3  -0.655969  0.193421
4   0.553439  1.318152
[5 rows x 2 columns]

The new functionality works well in method chains. Previously these methods only accepted functions or dicts mapping a label to a new label. This continues to work as before for function or dict-like values.

**Range Index**

A RangeIndex has been added to the Int64Index sub-classes to support a memory saving alternative for common use cases. This has a similar implementation to the python range object (xrange in python 2), in that it only stores the start, stop, and step values for the index. It will transparently interact with the user API, converting to Int64Index if needed.

This will now be the default constructed index for NDFrame objects, rather than previous an Int64Index. (GH939, GH12070, GH12071, GH12109, GH12888)

Previous behavior:

```
In [3]: s = pd.Series(range(1000))
In [4]: s.index
Out[4]:
Int64Index([  0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
...............  990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
dtype='int64', length=1000)

In [6]: s.index.nbytes
Out[6]: 8000
```

New behavior:

```
In [13]: s = pd.Series(range(1000))
In [14]: s.index
Out[14]: RangeIndex(start=0, stop=1000, step=1)
In [15]: s.index.nbytes
Out[15]: 128
```
Changes to `str.extract`

The `str.extract` method takes a regular expression with capture groups, finds the first match in each subject string, and returns the contents of the capture groups (GH11386).

In v0.18.0, the `expand` argument was added to `extract`.

- `expand=False`: it returns a Series, Index, or DataFrame, depending on the subject and regular expression pattern (same behavior as pre-0.18.0).
- `expand=True`: it always returns a DataFrame, which is more consistent and less confusing from the perspective of a user.

Currently the default is `expand=None` which gives a `FutureWarning` and uses `expand=False`. To avoid this warning, please explicitly specify `expand`.

```python
In [1]: pd.Series(["a1", "b2", "c3"]).str.extract(r"[ab](\d)", expand=None)
FutureWarning: currently extract(expand=None) means expand=False (return Index/Series/DataFrame)
but in a future version of pandas this will be changed to expand=True (return DataFrame)
Out[1]:
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>
```

Extracting a regular expression with one group returns a Series if `expand=False`.

```python
In [16]: pd.Series(["a1", "b2", "c3"]).str.extract(r"[ab](\d)", expand=False)
Out[16]:
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>
```

It returns a DataFrame with one column if `expand=True`.

```python
In [17]: pd.Series(["a1", "b2", "c3"]).str.extract(r"[ab](\d)", expand=True)
Out[17]:
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

Calling on an Index with a regex with exactly one capture group returns an Index if `expand=False`.

```python
In [18]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])
In [19]: s.index
Out[19]: Index(["A11", "B22", "C33"], dtype='object')
In [20]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[20]: Index(["A", "B", "C"], dtype='object', name='letter')
```
It returns a DataFrame with one column if `expand=True`.

```python
In [21]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[21]:
   letter
0   A
1   B
2   C
[3 rows x 1 columns]
```

Calling on an Index with a regex with more than one capture group raises `ValueError` if `expand=False`.

```python
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

It returns a DataFrame if `expand=True`.

```python
In [22]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[22]:
   letter  1
0   A   11
1   B   22
2   C   33
[3 rows x 2 columns]
```

In summary, `extract(expand=True)` always returns a DataFrame with a row for every subject string, and a column for every capture group.

**Addition of str.extractall**

The `.str.extractall` method was added (GH11386). Unlike `extract`, which returns only the first match.

```python
In [23]: s = pd.Series(["ala2", "b1", "c1"], ["A", "B", "C"])
In [24]: s
Out[24]:
A   a1
B   b1
C   c1
Length: 3, dtype: object
In [25]: s.str.extract(r"(?P<letter>[ab])(?P<digit>\d)"", expand=False)
Out[25]:
   letter  digit
A      a  1
B      b  1
C    NaN  NaN
[3 rows x 2 columns]
```

The `extractall` method returns all matches.
Changes to `str.cat`

The method `.str.cat()` concatenates the members of a `Series`. Before, if NaN values were present in the Series, calling `.str.cat()` on it would return NaN, unlike the rest of the `Series.str.*` API. This behavior has been amended to ignore NaN values by default. (GH11435).

A new, friendlier `ValueError` is added to protect against the mistake of supplying the `sep` as an arg, rather than as a kwarg. (GH11334).

```python
In [27]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ')
Out[27]: 'a b c'
```

```python
In [28]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ', na_rep='?')
Out[28]: 'a b ? c'
```

```python
In [2]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(' ')
ValueError: Did you mean to supply a `sep` keyword?
```

Datetimelike rounding

`DatetimeIndex`, `Timestamp`, `TimedeltaIndex`, `Timedelta` have gained the `.round()`, `.floor()` and `.ceil()` method for datetimelike rounding, flooring and ceiling. (GH4314, GH11963)

Naive datetimes

```python
In [29]: dr = pd.date_range('20130101 09:12:56.1234', periods=3)
```

```python
In [30]: dr
Out[30]:
DatetimeIndex(['2013-01-01 09:12:56.123400', '2013-01-02 09:12:56.123400',
               '2013-01-03 09:12:56.123400'],
              dtype='datetime64[ns]', freq='D')
```

```python
In [31]: dr.round('s')
Out[31]:
DatetimeIndex(['2013-01-01 09:12:56', '2013-01-02 09:12:56',
               '2013-01-03 09:12:56'],
              dtype='datetime64[ns]', freq=None)
```

```
# Timestamp scalar
In [32]: dr[0]
(continues on next page)
Out[32]: Timestamp('2013-01-01 09:12:56.123400', freq='D')

In [33]: dr[0].round('10s')
Out[33]: Timestamp('2013-01-01 09:13:00')

Tz-aware are rounded, floored and ceiled in local times

In [34]: dr = dr.tz_localize('US/Eastern')

In [35]: dr
Out[35]: DatetimeIndex(['2013-01-01 09:12:56-05:00',
                     '2013-01-02 09:12:56-05:00',
                     '2013-01-03 09:12:56-05:00'],
           dtype='datetime64[ns, US/Eastern]', freq=None)

In [36]: dr.round('s')
Out[36]: DatetimeIndex(['2013-01-01 09:12:56-05:00',
                      '2013-01-02 09:12:56-05:00',
                      '2013-01-03 09:12:56-05:00'],
                     dtype='datetime64[ns, US/Eastern]', freq=None)

Timedeltas

In [37]: t = pd.timedelta_range('1 days 2 hr 13 min 45 us', periods=3, freq='d')

In [38]: t
Out[38]: TimedeltaIndex(['1 days 02:13:00.000045', '2 days 02:13:00.000045',
                      '3 days 02:13:00.000045'],
                      dtype='timedelta64[ns]', freq='D')

In [39]: t.round('10min')
Out[39]: TimedeltaIndex(['1 days 02:10:00', '2 days 02:10:00', '3 days 02:10:00'],
                      dtype='timedelta64[ns]', freq=None)

# Timedelta scalar
In [40]: t[0]
Out[40]: Timedelta('1 days 02:13:00.000045')

In [41]: t[0].round('2h')
Out[41]: Timedelta('1 days 02:00:00')

In addition, .round(), .floor() and .ceil() will be available through the .dt accessor of Series.

In [42]: s = pd.Series(dr)

In [43]: s
Out[43]:
0  2013-01-01 09:12:56.123400-05:00
1  2013-01-02 09:12:56.123400-05:00
2  2013-01-03 09:12:56.123400-05:00
Length: 3, dtype: datetime64[ns, US/Eastern]
In [44]: s.dt.round('D')
Out[44]:
0  2013-01-01 00:00:00-05:00
1  2013-01-02 00:00:00-05:00
2  2013-01-03 00:00:00-05:00
Length: 3, dtype: datetime64[ns, US/Eastern]

Formatting of integers in FloatIndex

Integers in FloatIndex, e.g. 1., are now formatted with a decimal point and a 0 digit, e.g. 1.0 (GH11713) This change not only affects the display to the console, but also the output of IO methods like .to_csv or .to_html.

Previous behavior:

In [2]: s = pd.Series([1, 2, 3], index=np.arange(3.))
In [3]: s
Out[3]:
0 1
1 2
2 3
dtype: int64
In [4]: s.index
Out[4]: Float64Index([0.0, 1.0, 2.0], dtype='float64')

In [5]: print(s.to_csv(path=None))
0,1
1,2
2,3

New behavior:

In [45]: s = pd.Series([1, 2, 3], index=np.arange(3.))
In [46]: s
Out[46]:
0.0  1
1.0  2
2.0  3
Length: 3, dtype: int64
In [47]: s.index
Out[47]: Float64Index([0.0, 1.0, 2.0], dtype='float64')

In [48]: print(s.to_csv(path_or_buf=None, header=False))
0.0,1
1.0,2
2.0,3
Changes to dtype assignment behaviors

When a DataFrame’s slice is updated with a new slice of the same dtype, the dtype of the DataFrame will now remain the same. (GH10503)

Previous behavior:

```python
In [5]: df = pd.DataFrame({'a': [0, 1, 1],
                      'b': pd.Series([100, 200, 300], dtype='uint32')})

In [7]: df.dtypes
Out[7]:
a   int64
b   uint32
dtype: object

In [8]: ix = df['a'] == 1

In [9]: df.loc[ix, 'b'] = df.loc[ix, 'b']

In [11]: df.dtypes
Out[11]:
a   int64
b   uint32
Length: 2, dtype: object
```

New behavior:

```python
In [49]: df = pd.DataFrame({'a': [0, 1, 1],
                      ....:
                      'b': pd.Series([100, 200, 300], dtype='uint32')})

In [50]: df.dtypes
Out[50]:
a   int64
b   uint32
Length: 2, dtype: object

In [51]: ix = df['a'] == 1

In [52]: df.loc[ix, 'b'] = df.loc[ix, 'b']

In [53]: df.dtypes
Out[53]:
a   int64
b   uint32
Length: 2, dtype: object
```

When a DataFrame’s integer slice is partially updated with a new slice of floats that could potentially be down-casted to integer without losing precision, the dtype of the slice will be set to float instead of integer.

Previous behavior:

```python
In [4]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
```
In [5]: df
Out[5]:
   a b c  
4 8 1 2 3
10 4 5 6
8 12 7 8 9

In [7]: df.ix[4, 'c'] = np.array([0., 1.])

In [8]: df
Out[8]:
   a b c  
4 8 1 2 0
10 4 5 1
8 12 7 8 9

New behavior:

In [54]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
   ....:     columns=list('abc'),
   ....:     index=[[4,4,8], [8,10,12]])

In [55]: df
Out[55]:
   a b c  
4 8 1 2 3
10 4 5 6
8 12 7 8 9

[3 rows x 3 columns]

In [56]: df.loc[4, 'c'] = np.array([0., 1.])

In [57]: df
Out[57]:
   a b c  
4 8 1 2 0
10 4 5 1
8 12 7 8 9

[3 rows x 3 columns]
Method to_xarray

In a future version of pandas, we will be deprecating Panel and other > 2 ndim objects. In order to provide for continuity, all NDFrame objects have gained the `.to_xarray()` method in order to convert to xarray objects, which has a pandas-like interface for > 2 ndim. (GH11972)

See the xarray full-documentation here.

```
In [1]: p = Panel(np.arange(2*3*4).reshape(2,3,4))

In [2]: p.to_xarray()
Out[2]:
<xarray.DataArray (items: 2, major_axis: 3, minor_axis: 4)>
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
Coordinates:
   * items   (items) int64 0 1
   * major_axis (major_axis) int64 0 1 2
   * minor_axis (minor_axis) int64 0 1 2 3
```

Latex representation

DataFrame has gained a `.repr_latex()` method in order to allow for conversion to latex in a ipython/jupyter notebook using nbconvert. (GH11778)

Note that this must be activated by setting the option `pd.display.latex.repr=True` (GH12182)

For example, if you have a jupyter notebook you plan to convert to latex using nbconvert, place the statement `pd.display.latex.repr=True` in the first cell to have the contained DataFrame output also stored as latex.

The options `display.latex.escape` and `display.latex.longtable` have also been added to the configuration and are used automatically by the `to_latex` method. See the available options docs for more info.
pd.read_sas() changes

read_sas has gained the ability to read SAS7BDAT files, including compressed files. The files can be read in entirety, or incrementally. For full details see here. (GH4052)

Other enhancements

- Handle truncated floats in SAS xport files (GH11713)
- Added option to hide index in Series.to_string (GH11729)
- read_excel now supports s3 urls of the format s3://bucketname/filename (GH11447)
- add support for AWS_S3_HOST env variable when reading from s3 (GH12198)
- A simple version of Panel.round() is now implemented (GH11763)
- For Python 3.x, round(DataFrame), round(Series), round(Panel) will work (GH11763)
- sys.getsizeof(obj) returns the memory usage of a pandas object, including the values it contains (GH11597)
- Series gained an is_unique attribute (GH11946)
- DataFrame.quantile and Series.quantile now accept interpolation keyword (GH10174).
- Added DataFrame.style.format for more flexible formatting of cell values (GH11692)
- DataFrame.select_dtypes now allows the np.float16 type code (GH11990)
- pivot_table() now accepts most iterables for the values parameter (GH12017)
- Added Google BigQuery service account authentication support, which enables authentication on remote servers. (GH11881, GH12572). For further details see here
- HDFStore is now iterable: for k in store is equivalent to for k in store.keys() (GH12221).
- Add missing methods/fields to .dt for Period (GH8848)
- The entire code base has been PEP-ified (GH12096)

Backwards incompatible API changes

- the leading white spaces have been removed from the output of .to_string(index=False) method (GH11833)
- the out parameter has been removed from the Series.round() method. (GH11763)
- DataFrame.round() leaves non-numeric columns unchanged in its return, rather than raises. (GH11885)
- DataFrame.head(0) and DataFrame.tail(0) return empty frames, rather than self. (GH11937)
- Series.head(0) and Series.tail(0) return empty series, rather than self. (GH11937)
- to_msgpack and read_msgpack encoding now defaults to 'utf-8'. (GH12170)
- the order of keyword arguments to text file parsing functions (.read_csv(), .read_table(), .read_fwf()) changed to group related arguments. (GH11555)
- NaTType.isoformat now returns the string 'NaT to allow the result to be passed to the constructor of Timestamp. (GH12300)
**NaT and Timedelta operations**

NaT and Timedelta have expanded arithmetic operations, which are extended to Series arithmetic where applicable. Operations defined for datetime64[ns] or timedelta64[ns] are now also defined for NaT (GH11564).

NaT now supports arithmetic operations with integers and floats.

```
In [58]: pd.NaT * 1
Out[58]: NaT

In [59]: pd.NaT * 1.5
Out[59]: NaT

In [60]: pd.NaT / 2
Out[60]: NaT

In [61]: pd.NaT * np.nan
Out[61]: NaT
```

NaT defines more arithmetic operations with datetime64[ns] and timedelta64[ns].

```
In [62]: pd.NaT / pd.NaT
Out[62]: nan

In [63]: pd.Timedelta('1s') / pd.NaT
Out[63]: nan
```

NaT may represent either a datetime64[ns] null or a timedelta64[ns] null. Given the ambiguity, it is treated as a timedelta64[ns], which allows more operations to succeed.

```
In [64]: pd.NaT + pd.NaT
Out[64]: NaT

# same as
In [65]: pd.Timedelta('1s') + pd.Timedelta('1s')
Out[65]: Timedelta('0 days 00:00:02')
```

as opposed to

```
In [3]: pd.Timestamp('19900315') + pd.Timestamp('19900315')
TypeError: unsupported operand type(s) for +: 'Timestamp' and 'Timestamp'
```

However, when wrapped in a Series whose dtype is datetime64[ns] or timedelta64[ns], the dtype information is respected.

```
In [1]: pd.Series([pd.NaT], dtype='<M8[ns]') + pd.Series([pd.NaT], dtype='<M8[ns]')
TypeError: can only operate on a datetimes for subtraction, but the operator [__add__] was passed

In [66]: pd.Series([pd.NaT], dtype='<m8[ns]') + pd.Series([pd.NaT], dtype='<m8[ns]')
Out[66]:
    0   NaT
Length: 1, dtype: timedelta64[ns]
```

Timedelta division by floats now works.
Subtraction by Timedelta in a Series by a Timestamp works (GH11925)

NaT.isoformat() now returns 'NaT'. This change allows pd.Timestamp to rehydrate any timestamp like object from its isoformat (GH12300).

**Changes to msgpack**

Forward incompatible changes in msgpack writing format were made over 0.17.0 and 0.18.0; older versions of pandas cannot read files packed by newer versions (GH12129, GH10527)

Bugs in to_msgpack and read_msgpack introduced in 0.17.0 and fixed in 0.18.0, caused files packed in Python 2 unreadable by Python 3 (GH12142). The following table describes the backward and forward compat of msgpacks.

<table>
<thead>
<tr>
<th>Packed with</th>
<th>Can be unpacked with</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-0.17 / Python 2</td>
<td>any</td>
</tr>
<tr>
<td>pre-0.17 / Python 3</td>
<td>any</td>
</tr>
<tr>
<td>0.17 / Python 2</td>
<td>• ==0.17 / Python 2</td>
</tr>
<tr>
<td></td>
<td>• &gt;=0.18 / any Python</td>
</tr>
<tr>
<td>0.17 / Python 3</td>
<td>&gt;=0.18 / any Python</td>
</tr>
<tr>
<td>0.18</td>
<td>&gt;= 0.18</td>
</tr>
</tbody>
</table>

0.18.0 is backward-compatible for reading files packed by older versions, except for files packed with 0.17 in Python 2, in which case only they can only be unpacked in Python 2.
**Signature change for .rank**

Series.rank and DataFrame.rank now have the same signature (GH11759)

Previous signature

```python
In [3]: pd.Series([0,1]).rank(method='average', na_option='keep',
                      ascending=True, pct=False)
```

```text
Out[3]:
0 1
1 2
dtype: float64
```

```python
In [4]: pd.DataFrame([0,1]).rank(axis=0, numeric_only=None,
                           method='average', na_option='keep',
                           ascending=True, pct=False)
```

```text
Out[4]:
0 1
1 2
```

New signature

```python
In [71]: pd.Series([0,1]).rank(axis=0, method='average', numeric_only=False,
                               na_option='keep', ascending=True, pct=False)
```

```text
Out[71]:
0 1.0
1 2.0
Length: 2, dtype: float64
```

```python
In [72]: pd.DataFrame([0,1]).rank(axis=0, method='average', numeric_only=False,
                               na_option='keep', ascending=True, pct=False)
```

```text
Out[72]:
0 1.0
1 2.0
[2 rows x 1 columns]
```

**Bug in QuarterBegin with n=0**

In previous versions, the behavior of the QuarterBegin offset was inconsistent depending on the date when the n parameter was 0. (GH11406)

The general semantics of anchored offsets for n=0 is to not move the date when it is an anchor point (e.g., a quarter start date), and otherwise roll forward to the next anchor point.

```python
In [73]: d = pd.Timestamp('2014-02-01')

In [74]: d
Out[74]: Timestamp('2014-02-01 00:00:00')
```
In [75]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[75]: Timestamp('2014-02-01 00:00:00')

In [76]: d + pd.offsets.QuarterBegin(n=0, startingMonth=1)
Out[76]: Timestamp('2014-04-01 00:00:00')

For the QuarterBegin offset in previous versions, the date would be rolled backwards if date was in the same month as the quarter start date.

In [3]: d = pd.Timestamp('2014-02-15')

In [4]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[4]: Timestamp('2014-02-01 00:00:00')

This behavior has been corrected in version 0.18.0, which is consistent with other anchored offsets like MonthBegin and YearBegin.

In [77]: d = pd.Timestamp('2014-02-15')

In [78]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[78]: Timestamp('2014-05-01 00:00:00')

Resample API

Like the change in the window functions API above, .resample(...) is changing to have a more groupby-like API. (GH11732, GH12702, GH12202, GH12332, GH12334, GH12348, GH12448).

In [79]: np.random.seed(1234)

In [80]: df = pd.DataFrame(np.random.rand(10, 4),
....:           columns=list('ABCD'),
....:           index=pd.date_range('2010-01-01 09:00:00',
....:           periods=10, freq='s'))

In [81]: df
Out[81]:
          A         B         C         D
2010-01-01 09:00:00 0.191519 0.622109 0.437728 0.785359
2010-01-01 09:00:01 0.779976 0.272593 0.276464 0.801872
2010-01-01 09:00:02 0.958139 0.875933 0.357817 0.500995
2010-01-01 09:00:03 0.683463 0.712702 0.370251 0.561196
2010-01-01 09:00:04 0.503083 0.013768 0.772827 0.882641
2010-01-01 09:00:05 0.364886 0.615396 0.075381 0.368824
2010-01-01 09:00:06 0.933140 0.651378 0.397203 0.788730
2010-01-01 09:00:07 0.316836 0.568099 0.869127 0.436173
2010-01-01 09:00:08 0.802148 0.143767 0.704261 0.704581
2010-01-01 09:00:09 0.218792 0.924868 0.442141 0.909316

[10 rows x 4 columns]
Previous API:

You would write a resampling operation that immediately evaluates. If a `how` parameter was not provided, it would default to `how='mean'`.  

```python
In [6]: df.resample('2s')
Out[6]:
     A     B     C     D
2010-01-01 09:00:00 0.485748 0.447351 0.357096 0.793615
2010-01-01 09:00:02 0.820801 0.794317 0.364034 0.531096
2010-01-01 09:00:04 0.433985 0.314582 0.424104 0.625733
2010-01-01 09:00:06 0.624988 0.609738 0.633165 0.612452
2010-01-01 09:00:08 0.510470 0.534317 0.573201 0.806949
```

You could also specify a `how` directly

```python
In [7]: df.resample('2s', how='sum')
Out[7]:
     A     B     C     D
2010-01-01 09:00:00 0.971495 0.894701 0.714192 1.587231
2010-01-01 09:00:02 1.641602 1.588635 0.728068 1.062191
2010-01-01 09:00:04 0.867969 0.629165 0.848208 1.251465
2010-01-01 09:00:06 1.249976 1.219477 1.266330 1.224904
2010-01-01 09:00:08 1.020940 1.068634 1.146402 1.613897
```

New API:

Now, you can write `.resample(...)` as a 2-stage operation like `.groupby(...), which yields a `Resampler`. 

```python
In [82]: r = df.resample('2s')
In [83]: r
Out[83]: <pandas.core.resample.DatetimeIndexResampler object at 0x7f2d8ea71f10>
```

Downsampling

You can then use this object to perform operations. These are downsampling operations (going from a higher frequency to a lower one).

```python
In [84]: r.mean()
Out[84]:
     A     B     C     D
2010-01-01 09:00:00 0.485748 0.447351 0.357096 0.793615
2010-01-01 09:00:02 0.820801 0.794317 0.364034 0.531096
2010-01-01 09:00:04 0.433985 0.314582 0.424104 0.625733
2010-01-01 09:00:06 0.624988 0.609738 0.633165 0.612452
2010-01-01 09:00:08 0.510470 0.534317 0.573201 0.806949
[5 rows x 4 columns]
```

```python
In [85]: r.sum()
Out[85]:
     A     B     C     D
```

(continues on next page)
Furthermore, resample now supports `getitem` operations to perform the resample on specific columns.

```
In [86]: r[['A','C']].mean()
Out[86]:
          A         C
2010-01-01 09:00:00  0.485748  0.357096
2010-01-01 09:00:02  0.820801  0.364034
2010-01-01 09:00:04  0.433985  0.424104
2010-01-01 09:00:06  0.624988  0.633165
2010-01-01 09:00:08  0.510470  0.573201
```

and `.aggregate` type operations.

```
In [87]: r.agg({'A' : 'mean', 'B' : 'sum'})
Out[87]:
         A    B
2010-01-01 09:00:00 0.485748 0.894701
2010-01-01 09:00:02 0.820801 1.588635
2010-01-01 09:00:04 0.433985 0.629165
2010-01-01 09:00:06 0.624988 1.219477
2010-01-01 09:00:08 0.510470 1.068634
```

These accessors can of course, be combined

```
In [88]: r[['A','B']].agg(['mean','sum'])
Out[88]:
          A      B
         mean   sum  mean   sum
2010-01-01 09:00:00 0.485748 0.971495 0.447351 0.894701
2010-01-01 09:00:02 0.820801 1.641602 0.794317 1.588635
2010-01-01 09:00:04 0.433985 0.867969 0.314582 0.629165
2010-01-01 09:00:06 0.624988 1.249976 0.609738 1.219477
2010-01-01 09:00:08 0.510470 1.020940 0.534317 1.068634
```

[5 rows x 4 columns]
Upsampling

Upsampling operations take you from a lower frequency to a higher frequency. These are now performed with the Resampler objects with backfill(), ffill(), fillna() and asfreq() methods.

```python
In [89]: s = pd.Series(np.arange(5, dtype='int64'),
                  index=pd.date_range('2010-01-01', periods=5, freq='Q'))

In [90]: s
Out[90]:
2010-03-31  0
2010-06-30  1
2010-09-30  2
2010-12-31  3
2011-03-31  4
Freq: Q-DEC, Length: 5, dtype: int64
```

Previously

```python
In [6]: s.resample('M', fill_method='ffill')
Out[6]:
2010-03-31  0
2010-04-30  0
2010-05-31  0
2010-06-30  1
2010-07-31  1
2010-08-31  1
2010-09-30  2
2010-10-31  2
2010-11-30  2
2010-12-31  3
2011-01-31  3
2011-02-28  3
2011-03-31  4
Freq: M, dtype: int64
```

New API

```python
In [91]: s.resample('M').ffill()
Out[91]:
2010-03-31  0
2010-04-30  0
2010-05-31  0
2010-06-30  1
2010-07-31  1
2010-08-31  1
2010-09-30  2
2010-10-31  2
2010-11-30  2
2010-12-31  3
2011-01-31  3
2011-02-28  3
Freq: M, dtype: int64
```

(continues on next page)
Note: In the new API, you can either downsample OR upsample. The prior implementation would allow you to pass an aggregator function (like `mean`) even though you were upsampling, providing a bit of confusion.

 Previous API will work but with deprecations

**Warning:** This new API for resample includes some internal changes for the prior-to-0.18.0 API, to work with a deprecation warning in most cases, as the resample operation returns a deferred object. We can intercept operations and just do what the (pre 0.18.0) API did (with a warning). Here is a typical use case:

```
In [4]: r = df.resample('2s')

In [6]: r*10
```

```
pandas/tseries/resample.py:80: FutureWarning: .resample() is now a deferred operation
use .resample(...).mean() instead of .resample(...)  
```

```
Out[6]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>2010-01-01 09:00:00</td>
<td>4.857476</td>
<td>4.473507</td>
<td>3.570960</td>
<td>7.936154</td>
</tr>
<tr>
<td>2010-01-01 09:00:02</td>
<td>8.208011</td>
<td>7.943173</td>
<td>3.640340</td>
<td>5.310957</td>
</tr>
<tr>
<td>2010-01-01 09:00:04</td>
<td>4.339846</td>
<td>3.145823</td>
<td>4.241039</td>
<td>6.257326</td>
</tr>
<tr>
<td>2010-01-01 09:00:06</td>
<td>6.249881</td>
<td>6.097384</td>
<td>6.331650</td>
<td>6.124518</td>
</tr>
<tr>
<td>2010-01-01 09:00:08</td>
<td>5.104699</td>
<td>5.343172</td>
<td>5.732009</td>
<td>8.069486</td>
</tr>
</tbody>
</table>
```

However, getting and assignment operations directly on a Resampler will raise a `ValueError`:

```
In [7]: r.iloc[0] = 5

ValueError: .resample() is now a deferred operation
use .resample(...).mean() instead of .resample(...)  
```

There is a situation where the new API can not perform all the operations when using original code. This code is intending to resample every 2s, take the `mean` AND then take the `min` of those results.

```
In [4]: df.resample('2s').min()
```

```
Out[4]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.433985</td>
</tr>
<tr>
<td>B</td>
<td>0.314582</td>
</tr>
<tr>
<td>C</td>
<td>0.357096</td>
</tr>
<tr>
<td>D</td>
<td>0.531096</td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
</tr>
</tbody>
</table>
```

The new API will:

```
In [92]: df.resample('2s').min()
```

```
Out[92]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>2010-01-01 09:00:00</td>
<td>0.191519</td>
<td>0.272593</td>
<td>0.275293</td>
<td>0.785359</td>
</tr>
<tr>
<td>2010-01-01 09:00:02</td>
<td>0.683463</td>
<td>0.712702</td>
<td>0.357817</td>
<td>0.500995</td>
</tr>
<tr>
<td>2010-01-01 09:00:04</td>
<td>0.364886</td>
<td>0.013768</td>
<td>0.075381</td>
<td>0.368824</td>
</tr>
<tr>
<td>2010-01-01 09:00:06</td>
<td>0.316836</td>
<td>0.568099</td>
<td>0.397203</td>
<td>0.436173</td>
</tr>
<tr>
<td>2010-01-01 09:00:08</td>
<td>0.218792</td>
<td>0.143767</td>
<td>0.442141</td>
<td>0.704581</td>
</tr>
</tbody>
</table>
```

[5 rows x 4 columns]
The good news is the return dimensions will differ between the new API and the old API, so this should loudly raise an exception.

To replicate the original operation

```
In [93]: df.resample('2s').mean().min()
Out[93]:
   A    0.433985
   B    0.314582
   C    0.357096
   D    0.531096
Length: 4, dtype: float64
```

Changes to eval

In prior versions, new columns assignments in an `eval` expression resulted in an inplace change to the `DataFrame`. (GH9297, GH8664, GH10486)

```
In [94]: df = pd.DataFrame({'a': np.linspace(0, 10, 5), 'b': range(5)})
In [95]: df
Out[95]:
     a  b
   0  0.0  0
   1  2.5  1
   2  5.0  2
   3  7.5  3
   4 10.0  4
```

```
In [12]: df.eval('c = a + b')
```

FutureWarning: `eval` expressions containing an assignment currently default to operating `inplace`.
This will change in a future version of pandas, use `inplace=True` to avoid this warning.

```
In [13]: df
Out[13]:
     a  b  c
   0  0.0  0  0.0
   1  2.5  1  3.5
   2  5.0  2  7.0
   3  7.5  3 10.5
   4 10.0  4 14.0
```

In version 0.18.0, a new `inplace` keyword was added to choose whether the assignment should be done inplace or return a copy.

```
In [96]: df
Out[96]:
     a  b  c
   0  0.0  0  0.0
   1  2.5  1  3.5
   2  5.0  2  7.0
   3  7.5  3 10.5
   4 10.0  4 14.0
```

(continues on next page)
0 0.0 0 0.0
1 2.5 1 3.5
2 5.0 2 7.0
3 7.5 3 10.5
4 10.0 4 14.0

[5 rows x 3 columns]

In [97]: df.eval('d = c - b', inplace=False)
Out[97]:
   a  b  c  d
0  0  0  0  0
1  2.5 1  3.5  2.5
2  5.0 2  7.0  5.0
3  7.5 3 10.5  7.5
4 10.0 4 14.0 10.0

[5 rows x 4 columns]

In [98]: df
Out[98]:
   a  b  c
0  0.0 0.0
1  2.5 1  3.5
2  5.0 2  7.0
3  7.5 3 10.5
4 10.0 4 14.0

[5 rows x 3 columns]

In [99]: df.eval('d = c - b', inplace=True)

In [100]: df
Out[100]:
   a  b  c  d
0  0.0 0.0 0.0
1  2.5 1  3.5  2.5
2  5.0 2  7.0  5.0
3  7.5 3 10.5  7.5
4 10.0 4 14.0 10.0

[5 rows x 4 columns]

Warning: For backwards compatibility, inplace defaults to True if not specified. This will change in a future version of pandas. If your code depends on an inplace assignment you should update to explicitly set inplace=True.

The inplace keyword parameter was also added the query method.

In [101]: df.query('a > 5')
Out[101]:

(continues on next page)
In [102]: df.query('a > 5', inplace=True)

In [103]: df
Out[103]:
   a    b    c    d    e    f    g
0  3.0  7.5  3.0  10.5  7.5  15.0 -7.0
1  4.0  10.0  4.0  14.0  10.0  20.0 -2.0

[2 rows x 7 columns]

**Warning:** Note that the default value for `inplace` in a query is `False`, which is consistent with prior versions.

eval has also been updated to allow multi-line expressions for multiple assignments. These expressions will be evaluated one at a time in order. Only assignments are valid for multi-line expressions.
Other API changes

- DataFrame.between_time and Series.between_time now only parse a fixed set of time strings. Parsing of date strings is no longer supported and raises a ValueError. (GH11818)

```
In [107]: s = pd.Series(range(10), pd.date_range('2015-01-01', freq='H', periods=10))

In [108]: s.between_time("7:00am", "9:00am")
Out[108]:
2015-01-01 07:00:00    7
2015-01-01 08:00:00    8
2015-01-01 09:00:00    9
Freq: H, Length: 3, dtype: int64
```

This will now raise.

```
In [2]: s.between_time('20150101 07:00:00','20150101 09:00:00')
ValueError: Cannot convert arg ['20150101 07:00:00'] to a time.
```

- .memory_usage() now includes values in the index, as does memory_usage in .info() (GH11597)
- DataFrame.to_latex() now supports non-ascii encodings (eg utf-8) in Python 2 with the parameter encoding (GH7061)
- pandas.merge() and DataFrame.merge() will show a specific error message when trying to merge with an object that is not of type DataFrame or a subclass (GH12081)
- DataFrame.unstack and Series.unstack now take fill_value keyword to allow direct replacement of missing values when an unstack results in missing values in the resulting DataFrame. As an added benefit, specifying fill_value will preserve the data type of the original stacked data. (GH9746)
- As part of the new API for window functions and resampling, aggregation functions have been clarified, raising more informative error messages on invalid aggregations. (GH9052). A full set of examples are presented in groupby.
- Statistical functions for NDFrame objects (like sum(), mean(), min()) will now raise if non-numpy-compatible arguments are passed in for **kwargs (GH12301)
- .to_latex and .to_html gain a decimal parameter like .to_csv; the default is '. ' (GH12031)
- More helpful error message when constructing a DataFrame with empty data but with indices (GH8020)
- .describe() will now properly handle bool dtype as a categorical (GH6625)
- More helpful error message with an invalid .transform with user defined input (GH10165)
- Exponentially weighted functions now allow specifying alpha directly (GH10789) and raise ValueError if parameters violate $0 < \alpha \leq 1$ (GH12492)
Deprecations

- The functions `pd.rolling_*`, `pd.expanding_*`, and `pd.ewm*` are deprecated and replaced by the corresponding method call. Note that the new suggested syntax includes all of the arguments (even if default) (GH11603)

```
In [1]: s = pd.Series(range(3))

In [2]: pd.rolling_mean(s, window=2, min_periods=1)
   FutureWarning: pd.rolling_mean is deprecated for Series and will be removed in a future version, replace with Series.rolling(min_periods=1, window=2, center=False).mean()

Out[2]:
0  0.0
1  0.5
2  1.5
dtype: float64

In [3]: pd.rolling_cov(s, s, window=2)
   FutureWarning: pd.rolling_cov is deprecated for Series and will be removed in a future version, replace with Series.rolling(window=2).cov(other=<Series>)

Out[3]:
0   NaN
1  0.5
2  0.5
dtype: float64
```

- The `freq` and `how` arguments to the `.rolling`, `.expanding`, and `.ewm` (new) functions are deprecated, and will be removed in a future version. You can simply resample the input prior to creating a window function. (GH11603).

  For example, instead of `s.rolling(window=5, freq='D').max()` to get the max value on a rolling 5 Day window, one could use `s.resample('D').mean().rolling(window=5).max()`, which first resamples the data to daily data, then provides a rolling 5 day window.

- `pd.tseries.frequencies.get_offset_name` function is deprecated. Use offset's `.freqstr` property as alternative (GH11192)

- `pandas.stats.fama_macbeth` routines are deprecated and will be removed in a future version (GH6077)

- `pandas.stats.ols, pandas.stats.plm` and `pandas.stats.var` routines are deprecated and will be removed in a future version (GH6077)

- show a `FutureWarning` rather than a `DeprecationWarning` on using long-time deprecated syntax in HDFStore.select, where the `where` clause is not a string-like (GH12027)

- The `pandas.options.display.mpl_style` configuration has been deprecated and will be removed in a future version of pandas. This functionality is better handled by matplotlib’s `style sheets` (GH11783).
Removal of deprecated float indexers

In **GH4892** indexing with floating point numbers on a non-
Float64Index was deprecated (in version 0.14.0). In 0.18.0, this deprecation warning is removed and these will now raise a TypeError. (GH12165, GH12333)

```python
In [109]: s = pd.Series([1, 2, 3], index=[4, 5, 6])

In [110]: s
Out[110]:
4 1
5 2
6 3
Length: 3, dtype: int64

In [111]: s2 = pd.Series([1, 2, 3], index=list('abc'))

In [112]: s2
Out[112]:
a 1
b 2
c 3
Length: 3, dtype: int64

Previous behavior:

```python
# this is label indexing
In [2]: s[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
Out[2]: 2

# this is positional indexing
In [3]: s.iloc[1.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
Out[3]: 2

# this is label indexing
In [4]: s.loc[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
Out[4]: 2

# .ix would coerce 1.0 to the positional 1, and index
In [5]: s2.ix[1.0] = 10
FutureWarning: scalar indexers for index type Index should be integers and not floating point

In [6]: s2
Out[6]:
a 1
b 10
c 3
dtype: int64
```
New behavior:

For iloc, getting & setting via a float scalar will always raise.

```python
In [3]: s.iloc[2.0]
TypeError: cannot do label indexing on <class 'pandas.indexes.numeric.Int64Index'> with these indexers [2.0] of <type 'float'>
```

Other indexers will coerce to a like integer for both getting and setting. The FutureWarning has been dropped for .loc, .ix and [].

```python
In [113]: s[5.0]
Out[113]: 2

In [114]: s.loc[5.0]
Out[114]: 2
```

and setting

```python
In [115]: s_copy = s.copy()
In [116]: s_copy[5.0] = 10

In [117]: s_copy
Out[117]:
   4   1
   5  10
   6   3
Length: 3, dtype: int64
```

```python
In [118]: s_copy = s.copy()
In [119]: s_copy.loc[5.0] = 10

In [120]: s_copy
Out[120]:
   4   1
   5  10
   6   3
Length: 3, dtype: int64
```

Positional setting with .ix and a float indexer will ADD this value to the index, rather than previously setting the value by position.

```python
In [3]: s2.ix[1.0] = 10
In [4]: s2
Out[4]:
a    1
b    2
c    3
1.0  10
dtype: int64
```

Slicing will also coerce integer-like floats to integers for a non-Float64Index.
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [121]: s.loc[5.0:6]
Out[121]:
5  2
6  3
Length: 2, dtype: int64

Note that for floats that are NOT coercible to ints, the label based bounds will be excluded

In [122]: s.loc[5.1:6]
Out[122]:
6  3
Length: 1, dtype: int64

Float indexing on a Float64Index is unchanged.

In [123]: s = pd.Series([1, 2, 3], index=np.arange(3.))
In [124]: s[1.0]
Out[124]: 2
In [125]: s[1.0:2.5]
Out[125]:
1.0  2
2.0  3
Length: 2, dtype: int64

Removal of prior version deprecations/changes

- Removal of rolling_corr_pairwise in favor of .rolling().corr(pairwise=True) (GH4950)
- Removal of expanding_corr_pairwise in favor of .expanding().corr(pairwise=True) (GH4950)
- Removal of DataMatrix module. This was not imported into the pandas namespace in any event (GH12111)
- Removal of cols keyword in favor of subset in DataFrame.duplicated() and DataFrame. drop_duplicates() (GH6680)
- Removal of the read_frame and frame_query (both aliases for pd.read_sql) and write_frame (alias of to_sql) functions in the pd.io.sql namespace, deprecated since 0.14.0 (GH6292).
- Removal of the order keyword from .factorize() (GH6930)

Performance improvements

- Improved performance of andrews_curves (GH11534)
- Improved huge DatetimeIndex, PeriodIndex and TimedeltaIndex's ops performance including NaT (GH10277)
- Improved performance of pandas.concat (GH11958)
- Improved performance of StataReader (GH11591)
- Improved performance in construction of Categoricals with Series of datetimes containing NaT (GH12077)
- Improved performance of ISO 8601 date parsing for dates without separators (GH11899), leading zeros (GH11871) and with white space preceding the time zone (GH9714)
Bug fixes

- Bug in GroupBy.size when data-frame is empty. (GH11699)
- Bug in Period.end_time when a multiple of time period is requested (GH11738)
- Regression in .clip with tz-aware datetimes (GH11838)
- Bug in date_range when the boundaries fell on the frequency (GH11804, GH12409)
- Bug in consistency of passing nested dicts to .groupby(...).agg(...) (GH9052)
- Accept unicode in Timedelta constructor (GH11995)
- Bug in value label reading for StataReader when reading incrementally (GH12014)
- Bug in vectorized DateOffset when n parameter is 0 (GH11370)
- Compat for numpy 1.11 w.r.t. NaT comparison changes (GH12049)
- Bug in read_csv when reading from a StringIO in threads (GH11790)
- Bug in not treating NaT as a missing value in datetimelikes when factorizing & with Categoricals (GH12077)
- Bug in getitem when the values of a Series were tz-aware (GH12089)
- Bug in Series.str.get_dummies when one of the variables was 'name' (GH12180)
- Bug in pd.concat while concatenating tz-aware NaT series. (GH11693, GH11755, GH12217)
- Bug in pd.read_stata with version <= 108 files (GH11693)
- Bug in Series.resample using a frequency of Nano when the index is a DatetimeIndex and contains non-zero nanosecond parts (GH12037)
- Bug in resampling with .nunique and a sparse index (GH12352)
- Removed some compiler warnings (GH12471)
- Work around compat issues with boto in python 3.5 (GH11915)
- Bug in NaT subtraction from Timestamp or DatetimeIndex with timezones (GH11718)
- Bug in subtraction of Series of a single tz-aware Timestamp (GH12290)
- Use compat iterators in PY2 to support .next() (GH12299)
- Bug in Timedelta.round with negative values (GH11690)
- Bug in .loc against CategoricalIndex may result in normal Index (GH11586)
- Bug in DataFrame.info when duplicated column names exist (GH11761)
- Bug in .copy of datetime tz-aware objects (GH11794)
- Bug in Series.apply and Series.map where timedelta64 was not boxed (GH11349)
- Bug in DataFrame.set_index() with tz-aware Series (GH12358)
- Bug in subclasses of DataFrame where AttributeError did not propagate (GH11808)
- Bug groupby on tz-aware data where selection not returning Timestamp (GH11616)
- Bug in pd.read_clipboard and pd.to_clipboard functions not supporting Unicode; upgrade included pyperclip to v1.5.15 (GH9263)
- Bug in DataFrame.query containing an assignment (GH8664)
- Bug in from_msgpack where __contains__() fails for columns of the unpacked DataFrame, if the DataFrame has object columns. (GH11880)
• Bug in `.resample` on categorical data with TimedeltaIndex (GH12169)
• Bug in timezone info lost when broadcasting scalar datetime to DataFrame (GH11682)
• Bug in `Index` creation from Timestamp with mixed tz coerces to UTC (GH11488)
• Bug in parsing timezone offset strings with non-zero minutes (GH11708)
• Bug in `df.plot` using incorrect colors for bar plots under matplotlib 1.5+ (GH11614)
• Bug in the `groupby` plot method when using keyword arguments (GH11805).
• Bug in `DataFrame.duplicated` and `drop_duplicates` causing spurious matches when setting `keep=False` (GH11864)
• Bug in `.loc` result with duplicated key may have `Index` with incorrect dtype (GH11497)
• Bug in `pd.rolling_median` where memory allocation failed even with sufficient memory (GH11696)
• Bug in `DataFrame.style` with spurious zeros (GH12134)
• Bug in `DataFrame.style` with integer columns not starting at 0 (GH12125)
• Bug in `.style.bar` may not rendered properly using specific browser (GH11678)
• Bug in rich comparison of Timedelta with a numpy.array of Timedelta that caused an infinite recursion (GH11835)
• Bug in `DataFrame.round` dropping column index name (GH11986)
• Bug in `df.replace` while replacing value in mixed dtype `Dataframe` (GH11698)
• Bug in `Index` prevents copying name of passed `Index` when a new name is not provided (GH11973)
• Bug in `read_excel` failing to read any non-empty sheets when empty sheets exist and `sheetname=None` (GH11711)
• Bug in `read_excel` failing to raise `NotImplemented` error when keywords `parse_dates` and `date_parser` are provided (GH11544)
• Bug in `read_sql` with pymysql connections failing to return chunked data (GH11522)
• Bug in `.to_csv` ignoring formatting parameters `decimal, na_rep, float_format` for float indexes (GH11553)
• Bug in `Int64Index` and `Float64Index` preventing the use of the modulo operator (GH9244)
• Bug in `MultiIndex.drop` for not lexsorted MultiIndexes (GH12078)
• Bug in `DataFrame` when masking an empty `DataFrame` (GH11859)
• Bug in `.plot` potentially modifying the `colors` input when the number of columns didn’t match the number of series provided (GH12039).
• Bug in `Series.plot` failing when index has a `CustomBusinessDay` frequency (GH7222).
• Bug in `.to_sql` for `datetime.time` values with sqlite fallback (GH8341)
• Bug in `read_excel` failing to read data with one column when `squeeze=True` (GH12157)
• Bug in `read_excel` failing to read one empty column (GH12292, GH9002)
• Bug in `.groupby` where a `KeyError` was not raised for a wrong column if there was only one row in the dataframe (GH11741)
• Bug in `.read_csv` with dtype specified on empty data producing an error (GH12048)
• Bug in `.read_csv` where strings like `'2E'` are treated as valid floats (GH12237)
• Bug in building pandas with debugging symbols (GH12123)
• Removed millisecond property of DatetimeIndex. This would always raise a ValueError (GH12019).
• Bug in Series constructor with read-only data (GH11502)
• Removed pandas._testing.choice(). Should use np.random.choice(), instead. (GH12386)
• Bug in .loc setitem indexer preventing the use of a TZ-aware DatetimeIndex (GH12050)
• Bug in .style indexes and MultiIndexes not appearing (GH11655)
• Bug in to_msgpack and from_msgpack which did not correctly serialize or deserialize NaT (GH12307).
• Bug in .skew and .kurt due to roundoff error for highly similar values (GH11974)
• Bug in Timestamp constructor where microsecond resolution was lost if HHMMSS were not separated with ‘:’ (GH10041)
• Bug in buffer_rd_bytes src->buffer could be freed more than once if reading failed, causing a segfault (GH12098)
• Bug in crosstab where arguments with non-overlapping indexes would return a KeyError (GH10291)
• Bug in DataFrame.apply in which reduction was not being prevented for cases in which dtype was not a numpy dtype (GH12244)
• Bug when initializing categorical series with a scalar value. (GH12336)
• Bug when specifying a UTC DatetimeIndex by setting utc=True in .to_datetime (GH11934)
• Bug when increasing the buffer size of CSV reader in read_csv (GH12494)
• Bug when setting columns of a DataFrame with duplicate column names (GH12344)

Contributors

A total of 101 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• ARF +
• Alex Alekseyev +
• Andrew McPherson +
• Andrew Rosenfeld
• Andy Hayden
• Anthonios Partheniou
• Anton I. Sipos
• Ben +
• Ben North +
• Bran Yang +
• Chris
• Chris Carroux +
• Christopher C. Aycock +
• Christopher Scanlin +
• Cody +
• Da Wang +
• Daniel Grady +
• Dorozhko Anton +
• Dr-Irv +
• Erik M. Bray +
• Evan Wright
• Francis T. O’Donovan +
• Frank Cleary +
• Gianluca Rossi
• Graham Jeffries +
• Guillaume Horel
• Henry Hammond +
• Isaac Schwabacher +
• Jean-Mathieu Deschenes
• Jeff Reback
• Joe Jevnik +
• John Freeman +
• John Fremlin +
• Jonas Hoersch +
• Joris Van den Bossche
• Joris Vankerschaver
• Justin Lecher
• Justin Lin +
• Ka Wo Chen
• Keming Zhang +
• Kerby Shedden
• Kyle +
• Marco Farrugia +
• MasonGallo +
• MattRijk +
• Matthew Lurie +
• Maximilian Roos
• Mayank Asthana +
• Mortada Mehyar
• Moussa Taifi +
- Navreet Gill
- Nicolas Bonnotte
- Paul Reiners
- Philip Gura
- Pietro Battiston
- RahulHP
- Randy Carnevale
- Rinoc Johnson
- Rishipuri
- Sangmin Park
- Scott E Lasley
- Sereger13
- Shannon Wang
- Skipper Seabold
- Thierry Moisan
- Thomas A Caswell
- Toby Dylan Hocking
- Tom Augspurger
- Travis
- Trent Hauck
- Tux1
- Varun
- Wes McKinney
- Will Thompson
- Yoav Ram
- Yoong Kang Lim
- Yoshiki Vázquez Baeza
- Young Joong Kim
- Younggun Kim
- Yuval Langer
- alex argunov
- behzad nouri
- boombard
- brian-pantano
- chromy
- daniel
5.14 Version 0.17

5.14.1 Version 0.17.1 (November 21, 2015)

Note: We are proud to announce that pandas has become a sponsored project of the (NumFOCUS organization). This will help ensure the success of development of pandas as a world-class open-source project.

This is a minor bug-fix release from 0.17.0 and includes a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Support for Conditional HTML Formatting, see here
- Releasing the GIL on the csv reader & other ops, see here
- Fixed regression in DataFrame.drop_duplicates from 0.16.2, causing incorrect results on integer values (GH11376)

What's new in v0.17.1

- New features
  - Conditional HTML formatting
- Enhancements
- API changes
  - Deprecations
New features

Conditional HTML formatting

Warning: This is a new feature and is under active development. We'll be adding features an possibly making breaking changes in future releases. Feedback is welcome in GH11610

We’ve added experimental support for conditional HTML formatting: the visual styling of a DataFrame based on the data. The styling is accomplished with HTML and CSS. Accesses the styler class with the pandas.DataFrame.style, attribute, an instance of Styler with your data attached.

Here’s a quick example:

```python
In [1]: np.random.seed(123)
In [2]: df = pd.DataFrame(np.random.randn(10, 5), columns=list("abcde"))
In [3]: html = df.style.background_gradient(cmap="viridis", low=0.5)
```

We can render the HTML to get the following table.

Styler interacts nicely with the Jupyter Notebook. See the documentation for more.

Enhancements

- DatetimeIndex now supports conversion to strings with astype(str) (GH10442)
- Support for compression (gzip/bz2) in pandas.DataFrame.to_csv() (GH7615)
- pd.read_* functions can now also accept pathlib.Path, or py:py._path.local.LocalPath objects for the filepath_or_buffer argument. (GH11033) - The DataFrame and Series functions .to_csv(), .to_html() and .to_latex() can now handle paths beginning with tildes (e.g. ~/Documents/) (GH11438)
- DataFrame now uses the fields of a namedtuple as columns, if columns are not supplied (GH11181)
- DataFrame.iter tuples() now returns namedtuple objects, when possible. (GH11269, GH11625)
- Added axvlines_kwds to parallel coordinates plot (GH10709)
- Option to .info() and .memory_usage() to provide for deep introspection of memory consumption. Note that this can be expensive to compute and therefore is an optional parameter. (GH11595)

```python
In [4]: df = pd.DataFrame({"A": ["foo"] * 1000}) # noqa: F821
In [5]: df["B"] = df["A"].astype("category")
# shows the '+' as we have object dtypes
In [6]: df.info()
```
(continues on next page)
Index now has a `fillna` method (GH10089)

```python
In [8]: pd.Index([1, np.nan, 3]).fillna(2)
Out[8]: Float64Index([1.0, 2.0, 3.0], dtype='float64')
```

Series of type `category` now make `.str.<...>` and `.dt.<...>` accessor methods / properties available, if the categories are of that type. (GH10661)

```python
In [9]: s = pd.Series(list("aabb")).astype("category")
In [10]: s
Out[10]:
0  a
1  a
2  b
3  b
Length: 4, dtype: category
Categories (2, object): ['a', 'b']
In [11]: s.str.contains("a")
Out[11]:
0   True
1   True
2  False
3  False
Length: 4, dtype: bool
```

```python
In [12]: date = pd.Series(pd.date_range("1/1/2015", periods=5)).astype("category")
In [13]: date
```

(continues on next page)
Out[13]:
0    2015-01-01
1    2015-01-02
2    2015-01-03
3    2015-01-04
4    2015-01-05
Length: 5, dtype: category

In [14]: date.dt.day
Out[14]:
0    1
1    2
2    3
3    4
4    5
Length: 5, dtype: int64

- `pivot_table` now has a `margins_name` argument so you can use something other than the default of ‘All’ (GH3335)
- Implement export of `datetime64[ns, tz]` dtypes with a fixed HDF5 store (GH11411)
- Pretty printing sets (e.g. in DataFrame cells) now uses set literal syntax ({x, y}) instead of Legacy Python syntax (set([x, y])) (GH11215)
- Improve the error message in pandas.io.gbq.to_gbq() when a streaming insert fails (GH11285) and when the DataFrame does not match the schema of the destination table (GH11359)

**API changes**

- `raise NotImplementedError` in `Index.shift` for non-supported index types (GH8038)
- `min` and `max` reductions on `datetime64` and `timedelta64` typed series now result in `NaT` and not `nan` (GH11245).
- Indexing with a null key will raise a `TypeError`, instead of a `ValueError` (GH11356)
- `Series.ptp` will now ignore missing values by default (GH11163)

**Deprecations**

- The `pandas.io.ga` module which implements google-analytics support is deprecated and will be removed in a future version (GH11308)
- Deprecate the `engine` keyword in `.to_csv()`, which will be removed in a future version (GH11274)
Performance improvements

- Checking monotonic-ness before sorting on an index (GH11080)
- Series.dropna performance improvement when its dtype can’t contain NaN (GH11159)
- Release the GIL on most datetime field operations (e.g. DatetimeIndex.year, Series.dt.year), normalization, and conversion to and from Period, DatetimeIndex.to_period and PeriodIndex.to_timestamp (GH11263)
- Release the GIL on some rolling algos: rolling_median, rolling_mean, rolling_max, rolling_min, rolling_var, rolling_kurt, rolling_skew (GH11450)
- Release the GIL when reading and parsing text files in read_csv, read_table (GH11272)
- Improved performance of rolling_median (GH11450)
- Improved performance of to_excel (GH11352)
- Performance bug in repr of Categorical categories, which was rendering the strings before chopping them for display (GH11305)
- Performance improvement in Categorical.remove_unused_categories, (GH11643).
- Improved performance of Series constructor with no data and DatetimeIndex (GH11433)
- Improved performance of shift, cumprod, and cumsum with groupby (GH4095)

Bug fixes

- SparseArray.__iter__() now does not cause PendingDeprecationWarning in Python 3.5 (GH11622)
- Regression from 0.16.2 for output formatting of long floats/nan, restored in (GH11302)
- Series.sort_index() now correctly handles the inplace option (GH11402)
- Incorrectly distributed .c file in the build on PyPi when reading a csv of floats and passing na_values=<a scalar> would show an exception (GH11374)
- Bug in .to_latex() output broken when the index has a name (GH10660)
- Bug in HDFStore.append with strings whose encoded length exceeded the max unencoded length (GH11234)
- Bug in merging datetime64[ns, tz] dtypes (GH11405)
- Bug in HDFStore.select when comparing with a numpy scalar in a where clause (GH11283)
- Bug in using DataFrame.ix with a MultiIndex indexer (GH11372)
- Bug in date_range with ambiguous endpoints (GH11626)
- Prevent adding new attributes to the accessors .str, .dt and .cat. Retrieving such a value was not possible, so error out on setting it. (GH10673)
- Bug in tz-conversions with an ambiguous time and .dt accessors (GH11295)
- Bug in output formatting when using an index of ambiguous times (GH11619)
- Bug in comparisons of Series vs list-likes (GH11339)
- Bug in DataFrame.replace with a datetime64[ns, tz] and a non-compat to_replace (GH11326, GH11153)
- Bug in isnull where numpy.datetime64('NaT') in a numpy.array was not determined to be null(GH11206)
• Bug in list-like indexing with a mixed-integer Index (GH11320)
• Bug in pivot_table with margins=True when indexes are of Categorical dtype (GH10993)
• Bug in DataFrame.plot cannot use hex strings colors (GH10299)
• Regression in DataFrame.drop_duplicates from 0.16.2, causing incorrect results on integer values (GH11376)
• Bug in pd.eval where unary ops in a list error (GH11235)
• Bug in squeeze() with zero length arrays (GH11230, GH8999)
• Bug in describe() dropping column names for hierarchical indexes (GH11517)
• Bug in DataFrame.pct_change() not propagating axis keyword on .fillna method (GH11150)
• Bug in .to_csv() when a mix of integer and string column names are passed as the columns parameter (GH11637)
• Bug in indexing with a range, (GH11652)
• Bug in inference of numpy scalars and preserving dtype when setting columns (GH11638)
• Bug in to_sql using unicode column names giving UnicodeEncodeError with (GH11431).
• Fix regression in setting of xticks in plot (GH11529).
• Bug in holiday.dates where observance rules could not be applied to holiday and doc enhancement (GH11477, GH11533)
• Fix plotting issues when having plain Axes instances instead of SubplotAxes (GH11520, GH11556).
• Bug in DataFrame.to_latex() produces an extra rule when header=False (GH11327)
• Bug in DataFrame.corr() raises exception when computes Kendall correlation for DataFrames with boolean and not boolean columns (GH11560)
• Bug in the link-time error caused by C inline functions on FreeBSD 10+ (with clang) (GH10510)
• Bug in DataFrame.to_csv in passing through arguments for formatting MultiIndexes, including date_format (GH7791)
• Bug in DataFrame.join() with how='right' producing a TypeError (GH11519)
• Bug in Series.quantile with empty list results has Index with object dtype (GH11588)
• Bug in pd.merge results in empty Int64Index rather than Index(dtype=object) when the merge result is empty (GH11588)
• Bug in Categorical.remove_unused_categories when having NaN values (GH11599)
• Bug in DataFrame.to_sparse() loses column names for MultiIndex (GH11600)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in `DataFrame.round()` with non-unique column index producing a Fatal Python error (GH11611)
- Bug in `DataFrame.round()` with `decimals` being a non-unique indexed Series producing extra columns (GH11618)

Contributors

A total of 63 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aleksandr Drozd +
- Alex Chase +
- Anthonios Partheniou
- BrenBarn +
- Brian J. McGuirk +
- Chris
- Christian Berendt +
- Christian Perez +
- Cody Piersall +
- Data & Code Expert Experimenting with Code on Data
- DrIrV +
- Evan Wright
- Guillaume Gay
- Hamed Saljooghinejad +
- Iblis Lin +
- Jake VanderPlas
- Jan Schulz
- Jean-Mathieu Deschenes +
- Jeff Reback
- Jimmy Callin +
- Joris Van den Bossche
- K.-Michael Aye
- Ka Wo Chen
- Loïc Séguin-C +
- Luo Yicheng +
- Magnus Jöud +
- Manuel Leonhardt +
- Matthew Gilbert
- Maximilian Roos
- Michael +
• Nicholas Stahl +
• Nicolas Bonnotte +
• Pastafarianist +
• Petra Chong +
• Phil Schaf +
• Philipp A +
• Rob deCarvalho +
• Roman Khomenko +
• Rémy Léone +
• Sebastian Bank +
• Sinhrks
• Stephan Hoyer
• Thierry Moisan
• Tom Augspurger
• Tux1 +
• Varun +
• Wieland Hoffmann +
• Winterflower
• Yoav Ram +
• Younggun Kim
• Zeke +
• ajcr
• azuranski +
• behzad nouri
• cel4
• emilydolson +
• hironow +
• lexical
• lllllllllll +
• rockg
• silentquasar +
• sinhrks
• taeold +
5.14.2 Version 0.17.0 (October 9, 2015)

This is a major release from 0.16.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

**Warning:** pandas >= 0.17.0 will no longer support compatibility with Python version 3.2 (GH9118)

**Warning:** The pandas.io.data package is deprecated and will be replaced by the pandas-datareader package. This will allow the data modules to be independently updated to your pandas installation. The API for pandas-datareader v0.1.1 is exactly the same as in pandas v0.17.0 (GH8961, GH10861).

After installing pandas-datareader, you can easily change your imports:

```python
from pandas.io import data, wb
```

becomes

```python
from pandas_datareader import data, wb
```

Highlights include:

- Release the Global Interpreter Lock (GIL) on some cython operations, see [here](#)
- Plotting methods are now available as attributes of the `.plot` accessor, see [here](#)
- The sorting API has been revamped to remove some long-time inconsistencies, see [here](#)
- Support for a `datetime64[ns]` with timezones as a first-class dtype, see [here](#)
- The default for `to_datetime` will now be to `raise` when presented with unparsable formats, previously this would return the original input. Also, date parse functions now return consistent results. See [here](#)
- The default for `dropna` in HDFStore has changed to `False`, to store by default all rows even if they are all NaN, see [here](#)
- Datetime accessor (`dt`) now supports `Series.dt.strftime` to generate formatted strings for datetime-likes, and `Series.dt.total_seconds` to generate each duration of the timedelta in seconds. See [here](#)
- Period and PeriodIndex can handle multiplied freq like 3D, which corresponding to 3 days span. See [here](#)
- Development installed versions of pandas will now have PEP440 compliant version strings (GH9518)
- Development support for benchmarking with the Air Speed Velocity library (GH8361)
- Support for reading SAS xport files, see [here](#)
- Documentation comparing SAS to pandas, see [here](#)
- Removal of the automatic TimeSeries broadcasting, deprecated since 0.8.0, see [here](#)
- Display format with plain text can optionally align with Unicode East Asian Width, see [here](#)
- Compatibility with Python 3.5 (GH11097)
- Compatibility with matplotlib 1.5.0 (GH11111)

Check the [API Changes](#) and [deprecations](#) before updating.
What’s new in v0.17.0

- **New features**
  - Datetime with TZ
  - Releasing the GIL
  - Plot submethods
  - Additional methods for dt accessor
    * Series.dt.strftime
    * Series.dt.total_seconds
  - Period frequency enhancement
  - Support for SAS XPORT files
  - Support for math functions in .eval()
  - Changes to Excel with MultiIndex
  - Google BigQuery enhancements
  - Display alignment with Unicode East Asian width
  - Other enhancements

- **Backwards incompatible API changes**
  - Changes to sorting API
  - Changes to to_datetime and to_timedelta
    * Error handling
    * Consistent parsing
  - Changes to Index comparisons
  - Changes to boolean comparisons vs. None
  - HDFStore dropna behavior
  - Changes to display.precision option
  - Changes to Categorical.unique
  - Changes to bool passed as header in parsers
  - Other API changes
  - Deprecations
  - Removal of prior version deprecations/changes

- **Performance improvements**

- **Bug fixes**

- **Contributors**
New features

Datetime with TZ

We are adding an implementation that natively supports datetime with timezones. A Series or a DataFrame column previously could be assigned a datetime with timezones, and would work as an object dtype. This had performance issues with a large number rows. See the docs for more details. (GH8260, GH10763, GH11034).

The new implementation allows for having a single-timezone across all rows, with operations in a performant manner.

```
In [1]: df = pd.DataFrame(
   ...:     {
   ...:         "A": pd.date_range("20130101", periods=3),
   ...:         "B": pd.date_range("20130101", periods=3, tz="US/Eastern"),
   ...:         "C": pd.date_range("20130101", periods=3, tz="CET"),
   ...:     }
   ...: )

In [2]: df
Out[2]:
  A         B         C
0 2013-01-01 00:00:00  00:00:00-05:00
1 2013-01-02 00:00:00  00:00:00-05:00
2 2013-01-03 00:00:00  00:00:00-05:00

[3 rows x 3 columns]
```

```
In [3]: df.dtypes
Out[3]:
   A    datetime64[ns]
  B  datetime64[ns, US/Eastern]
  C  datetime64[ns, CET]
Length: 3, dtype: object
```

```
In [4]: df.B
Out[4]:
0 2013-01-01 00:00:00-05:00
1 2013-01-02 00:00:00-05:00
2 2013-01-03 00:00:00-05:00
Name: B, Length: 3, dtype: datetime64[ns, US/Eastern]

In [5]: df.B.dt.tz_localize(None)
Out[5]:
0 2013-01-01
1 2013-01-02
2 2013-01-03
Name: B, Length: 3, dtype: datetime64[ns]
```

This uses a new-dtype representation as well, that is very similar in look-and-feel to its numpy cousin datetime64[ns]

```
In [6]: df["B"].dtype
Out[6]: datetime64[ns, US/Eastern]
```
In [7]: type(df["B"].dtype)
Out[7]: pandas.core.dtypes.dtypes.DatetimeTZDtype

Note: There is a slightly different string repr for the underlying DatetimeIndex as a result of the dtype changes, but functionally these are the same.

Previous behavior:

In [1]: pd.date_range('20130101', periods=3, tz='US/Eastern')
Out[1]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00', '2013-01-03 00:00:00-05:00'], dtype='datetime64[ns]', freq='D', tz='US/Eastern')

In [2]: pd.date_range('20130101', periods=3, tz='US/Eastern').dtype
Out[2]: dtype('<M8[ns]')

New behavior:

In [8]: pd.date_range("20130101", periods=3, tz="US/Eastern")
Out[8]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00', '2013-01-03 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]', freq='D')

In [9]: pd.date_range("20130101", periods=3, tz="US/Eastern").dtype
Out[9]: datetime64[ns, US/Eastern]

Releasing the GIL

We are releasing the global-interpreter-lock (GIL) on some cython operations. This will allow other threads to run simultaneously during computation, potentially allowing performance improvements from multi-threading. Notably groupby, nsmallest, value_counts and some indexing operations benefit from this. (GH8882)

For example the groupby expression in the following code will have the GIL released during the factorization step, e.g. df.groupby("key") as well as the .sum() operation.

N = 1000000
ngroups = 10
df = DataFrame(
    {"key": np.random.randint(0, ngroups, size=N), "data": np.random.randn(N)}
)
df.groupby("key")["data"].sum()

Releasing of the GIL could benefit an application that uses threads for user interactions (e.g. QT), or performing multi-threaded computations. A nice example of a library that can handle these types of computation-in-parallel is the dask library.
Plot submethods

The Series and DataFrame .plot() method allows for customizing *plot types* by supplying the kind keyword arguments. Unfortunately, many of these kinds of plots use different required and optional keyword arguments, which makes it difficult to discover what any given plot kind uses out of the dozens of possible arguments.

To alleviate this issue, we have added a new, optional plotting interface, which exposes each kind of plot as a method of the .plot attribute. Instead of writing `series.plot(kind=<kind>, ...)`, you can now also use `series.plot.<kind>(...)`:

```python
In [10]: df = pd.DataFrame(np.random.rand(10, 2), columns=['a', 'b'])
In [11]: df.plot.bar()
```

As a result of this change, these methods are now all discoverable via tab-completion:

```python
In [12]: df.plot.<TAB> # noqa: E225, E999
df.plot.area  df.plot.barh  df.plot.density  df.plot.hist  df.plot.line  df.plot.scatter  
df.plot.bar  df.plot.box  df.plot.hexbin  df.plot.kde  df.plot.pie
```

Each method signature only includes relevant arguments. Currently, these are limited to required arguments, but in the future these will include optional arguments, as well. For an overview, see the new *Plotting* API documentation.

**Additional methods for dt accessor**

**Series.dt.strftime**

We are now supporting a `Series.dt.strftime` method for datetime-likes to generate a formatted string *(GH10110).* Examples:

```python
# DatetimeIndex
In [13]: s = pd.Series(pd.date_range("20130101", periods=4))
In [14]: s
Out[14]:
```

(continues on next page)
0  2013-01-01
1  2013-01-02
2  2013-01-03
3  2013-01-04
Length: 4, dtype: datetime64[ns]

In [15]: s.dt.strftime("%Y/%m/%d")
Out[15]:
0   2013/01/01
1   2013/01/02
2   2013/01/03
3   2013/01/04
Length: 4, dtype: object

# PeriodIndex
In [16]: s = pd.Series(pd.period_range("20130101", periods=4))

In [17]: s
Out[17]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
Length: 4, dtype: period[D]

In [18]: s.dt.strftime("%Y/%m/%d")
Out[18]:
0   2013/01/01
1   2013/01/02
2   2013/01/03
3   2013/01/04
Length: 4, dtype: object

The string format is as the python standard library and details can be found here

**Series.dt.total_seconds**

pd.Series of type timedelta64 has new method .dt.total_seconds() returning the duration of the timedelta in seconds (GH10817)

# TimedeltaIndex
In [19]: s = pd.Series(pd.timedelta_range("1 minutes", periods=4))

In [20]: s
Out[20]:
0   0 days 00:01:00
1   1 days 00:01:00
2   2 days 00:01:00
3   3 days 00:01:00
Length: 4, dtype: timedelta64[ns]
In [21]: s.dt.total_seconds()
Out[21]:
0   60.0
1  86460.0
2 172860.0
3 259260.0
Length: 4, dtype: float64

Period frequency enhancement

Period, PeriodIndex and period_range can now accept multiplied freq. Also, Period.freq and PeriodIndex.freq are now stored as a DateOffset instance like DatetimeIndex, and not as str (GH7811)

A multiplied freq represents a span of corresponding length. The example below creates a period of 3 days. Addition and subtraction will shift the period by its span.

In [22]: p = pd.Period("2015-08-01", freq="3D")
In [23]: p
Out[23]: Period('2015-08-01', '3D')
In [24]: p + 1
Out[24]: Period('2015-08-04', '3D')
In [25]: p - 2
Out[25]: Period('2015-07-26', '3D')
In [26]: p.to_timestamp()
Out[26]: Timestamp('2015-08-01 00:00:00')
In [27]: p.to_timestamp(how="E")
Out[27]: Timestamp('2015-08-03 23:59:59.999999999')

You can use the multiplied freq in PeriodIndex and period_range.

In [28]: idx = pd.period_range("2015-08-01", periods=4, freq="2D")

In [29]: idx
Out[29]: PeriodIndex(['2015-08-01', '2015-08-03', '2015-08-05', '2015-08-07'], dtype='period[2D]')

In [30]: idx + 1
Out[30]: PeriodIndex(['2015-08-03', '2015-08-05', '2015-08-07', '2015-08-09'], dtype='period[2D]')
Support for SAS XPORT files

`read_sas()` provides support for reading *SAS XPORT* format files. (GH4052).

```python
df = pd.read_sas("sas_xport.xpt")
```

It is also possible to obtain an iterator and read an XPORT file incrementally.

```python
for df in pd.read_sas("sas_xport.xpt", chunksize=10000):
    do_something(df)
```

See the docs for more details.

Support for math functions in `.eval()`

`.eval()` now supports calling math functions (GH4893)

```python
def = pd.DataFrame({"a": np.random.randn(10)})
def.eval("b = sin(a)")
```

The support math functions are sin, cos, exp, log, expm1, log1p, sqrt, sinh, cosh, tanh, arcsin, arccos, arctan, arccosh, arcsinh, arctanh, abs and arctan2.

These functions map to the intrinsics for the NumExpr engine. For the Python engine, they are mapped to NumPy calls.

Changes to Excel with MultiIndex

In version 0.16.2 a DataFrame with MultiIndex columns could not be written to Excel via `to_excel`. That functionality has been added (GH10564), along with updating `read_excel` so that the data can be read back with, no loss of information, by specifying which columns/rows make up the MultiIndex in the header and `index_col` parameters (GH4679)

See the documentation for more details.

```python
In [31]: df = pd.DataFrame(
    ....:     [[1, 2, 3, 4], [5, 6, 7, 8]],
    ....:     columns=pd.MultiIndex.from_product(
    ....:         [["foo", "bar"], ["a", "b"]], names=["col1", "col2"]
    ....:     ),
    ....:     index=pd.MultiIndex.from_product([["j"], ["l", "k"]], names=["i1", "i2"]),
    ....: )
```

```python
In [32]: df
Out[32]:
   col1  foo  bar
  col2    a    b
   j  l  1  2  3  4
   k  5  6  7  8
[2 rows x 4 columns]
```
Previously, it was necessary to specify the `has_index_names` argument in `read_excel`, if the serialized data had index names. For version 0.17.0 the output format of `to_excel` has been changed to make this keyword unnecessary - the change is shown below.

**Old**

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>idx_name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2000-01-07 00:00:00</td>
<td>0.968129</td>
<td>0.906529</td>
<td>0.05343</td>
<td>0.02619</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2000-01-10 00:00:00</td>
<td>-0.16632</td>
<td>1.981993</td>
<td>1.833093</td>
<td>0.803685</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2000-01-11 00:00:00</td>
<td>0.121057</td>
<td>0.36946</td>
<td>-0.02888</td>
<td>1.683975</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2000-01-12 00:00:00</td>
<td>-1.70456</td>
<td>-0.73098</td>
<td>-0.38088</td>
<td>0.020946</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2000-01-13 00:00:00</td>
<td>-1.20024</td>
<td>1.907733</td>
<td>0.629318</td>
<td>1.507033</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2000-01-14 00:00:00</td>
<td>-0.66344</td>
<td>0.073188</td>
<td>1.583482</td>
<td>0.735205</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2000-01-17 00:00:00</td>
<td>0.716635</td>
<td>-2.07952</td>
<td>1.760536</td>
<td>0.970309</td>
<td></td>
</tr>
</tbody>
</table>
```

**New**

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>idx_name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2000-01-07 00:00:00</td>
<td>0.968129</td>
<td>0.906529</td>
<td>0.05343</td>
<td>0.02619</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2000-01-10 00:00:00</td>
<td>-0.16632</td>
<td>1.981993</td>
<td>1.833093</td>
<td>0.803685</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2000-01-11 00:00:00</td>
<td>0.121057</td>
<td>0.36946</td>
<td>-0.02888</td>
<td>1.683975</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2000-01-12 00:00:00</td>
<td>-1.70456</td>
<td>-0.73098</td>
<td>-0.38088</td>
<td>0.020946</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2000-01-13 00:00:00</td>
<td>-1.20024</td>
<td>1.907733</td>
<td>0.629318</td>
<td>1.507033</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2000-01-14 00:00:00</td>
<td>-0.66344</td>
<td>0.073188</td>
<td>1.583482</td>
<td>0.735205</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2000-01-17 00:00:00</td>
<td>0.716635</td>
<td>-2.07952</td>
<td>1.760536</td>
<td>0.970309</td>
<td></td>
</tr>
</tbody>
</table>
```

**Warning:** Excel files saved in version 0.16.2 or prior that had index names will still able to be read in, but the `has_index_names` argument must specified to True.
Google BigQuery enhancements

- Added ability to automatically create a table/dataset using the pandas.io.gbq.to_gbq() function if the destination table/dataset does not exist. (GH8325, GH11121).
- Added ability to replace an existing table and schema when calling the pandas.io.gbq.to_gbq() function via the if_exists argument. See the docs for more details (GH8325).
- InvalidColumnOrder and InvalidPageToken in the gbq module will raise ValueError instead of IOError.
- The generate_bq_schema() function is now deprecated and will be removed in a future version (GH11121)
- The gbq module will now support Python 3 (GH11094).

Display alignment with Unicode East Asian width

**Warning:** Enabling this option will affect the performance for printing of DataFrame and Series (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters its width is corresponding to 2 alphabets. If a DataFrame or Series contains these characters, the default output cannot be aligned properly. The following options are added to enable precise handling for these characters.

- display.unicode.east_asian_width: Whether to use the Unicode East Asian Width to calculate the display text width. (GH2612)
- display.unicode.ambiguous_as_wide: Whether to handle Unicode characters belong to Ambiguous as Wide. (GH11102)

```python
In [36]: df = pd.DataFrame({"UK": ["UK", ""], "":" ["Alice", ""]})
```

```python
In [37]: df
Out[37]:
0  UK  Alice
1
[2 rows x 2 columns]
```

```python
In [38]: pd.set_option("display.unicode.east_asian_width", True)
```

```python
In [39]: df
Out[39]:
0  UK  Alice
1
[2 rows x 2 columns]
```

For further details, see [here](#).
### Other enhancements

- Support for `openpyxl >= 2.2`. The API for style support is now stable (GH10125)
- `merge` now accepts the argument `indicator` which adds a Categorical-type column (by default called `_merge`) to the output object that takes on the values (GH8790)

<table>
<thead>
<tr>
<th>Observation Origin</th>
<th>_merge value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge key only in 'left' frame</td>
<td>left_only</td>
</tr>
<tr>
<td>Merge key only in 'right' frame</td>
<td>right_only</td>
</tr>
<tr>
<td>Merge key in both frames</td>
<td>both</td>
</tr>
</tbody>
</table>

```
In [40]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})
In [41]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})
In [42]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[42]:
          col1  col_left  col_right  _merge
0         0      a      NaN  left_only
1         1      b      2.0      both
2         2  NaN      2.0  right_only
3         2  NaN      2.0  right_only
```

For more, see the [updated docs](https://pandas.pydata.org/docs/).

- `pd.to_numeric` is a new function to coerce strings to numbers (possibly with coercion) (GH11133)
- `pd.merge` will now allow duplicate column names if they are not merged upon (GH10639).
- `pd.pivot` will now allow passing index as `None` (GH3962).
- `pd.concat` will now use existing Series names if provided (GH10698).

```
In [43]: foo = pd.Series([1, 2], name="foo")
In [44]: bar = pd.Series([1, 2])
In [45]: baz = pd.Series([4, 5])
```

Previous behavior:

```
In [1]: pd.concat([foo, bar, baz], axis=1)
Out[1]:
     0  1
   0  1  4
   1  2  5
```

New behavior:

```
In [46]: pd.concat([foo, bar, baz], axis=1)
Out[46]:
     foo  0  1
```

(continues on next page)
DataFrame has gained the nlargest and nsmallest methods (GH10393)

- Add a limit_direction keyword argument that works with limit to enable interpolate to fill NaN values forward, backward, or both (GH9218, GH10420, GH11115)

```
In [47]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13])

In [48]: ser.interpolate(limit=1, limit_direction="both")
Out[48]:
 0    NaN
 1    5.0
 2    5.0
 3    7.0
 4    NaN
 5   11.0
 6   13.0
Length: 7, dtype: float64
```

- Added a DataFrame.round method to round the values to a variable number of decimal places (GH10568).

```
In [49]: df = pd.DataFrame(
    ....:     np.random.random([3, 3]),
    ....:     columns=["A", "B", "C"],
    ....:     index=["first", "second", "third"],
    ....: )

In [50]: df.round(2)
Out[50]:
        A     B     C
first  0.13  0.97  0.26
second 0.90  0.38  0.34
third  0.45  0.84  0.12
```

```
In [51]: df.round({"A": 0, "C": 2})
Out[51]:
        A     B     C
first  0  0.97  0.26
second 0.90  0.38  0.34
third  0.45  0.84  0.12
```

(continues on next page)
first  0.0  0.966718  0.26  
second  1.0  0.376750  0.34  
third  0.0  0.840255  0.12  
[3 rows x 3 columns]

• **drop_duplicates** and **duplicated** now accept a keep keyword to target first, last, and all duplicates. The **take_last** keyword is deprecated, see [here](GH6511, GH8505)

```
In [53]: s = pd.Series(['A', 'B', 'C', 'A', 'B', 'D'])
In [54]: s.drop_duplicates()
Out[54]:
0    A
1    B
2    C
3    A
5    D
Length: 4, dtype: object
In [55]: s.drop_duplicates(keep='last')
Out[55]:
2    C
3    A
4    B
5    D
Length: 4, dtype: object
In [56]: s.drop_duplicates(keep=False)
Out[56]:
2    C
5    D
Length: 2, dtype: object
```

• **Reindex** now has a **tolerance** argument that allows for finer control of **Limits on filling while reindexing** (GH10411):

```
In [57]: df = pd.DataFrame({'x': range(5), 't': pd.date_range('2000-01-01', periods=5)})
In [58]: df.reindex([0.1, 1.9, 3.5], method='nearest', tolerance=0.2)
Out[58]:
   x       t
0.1  0.0  2000-01-01
1.9  2.0  2000-01-03
3.5  NaN    NaT
[3 rows x 2 columns]
```

When used on a **DatetimeIndex**, **TimedeltaIndex** or **PeriodIndex**, **tolerance** will coerced into a **Timedelta** if possible. This allows you to specify tolerance with a string:

```
In [59]: df = df.set_index('t')
```

(continues on next page)
In [60]: df.reindex(pd.to_datetime(["1999-12-31"], method="nearest", tolerance="1\n˓day")
Out[60]:

1999-12-31 0
[1 rows x 1 columns]

tolerance is also exposed by the lower level Index.get_indexer and Index.get_loc methods.

• Added functionality to use the base argument when resampling a TimeDeltaIndex (GH10530)
• DatetimeIndex can be instantiated using strings contains NaT (GH7599)
• to_datetime can now accept the yearfirst keyword (GH7599)
• pandas.tseries.offsets larger than the Day offset can now be used with a Series for addition/subtraction (GH10699). See the docs for more details.
• pd.Timedelta.total_seconds() now returns Timedelta duration to ns precision (previously microsecond precision) (GH10939)
• PeriodIndex now supports arithmetic with np.ndarray (GH10638)
• Support pickling of Period objects (GH10439)
• .as_blocks will now take a copy optional argument to return a copy of the data, default is to copy (no change in behavior from prior versions), (GH9607)
• regex argument to DataFrame.filter now handles numeric column names instead of raising ValueError (GH10384).
• Enable reading gzip compressed files via URL, either by explicitly setting the compression parameter or by inferring from the presence of the HTTP Content-Encoding header in the response (GH8685)
• Enable writing Excel files in memory using StringIO/BytesIO (GH7074)
• Enable serialization of lists and dicts to strings in ExcelWriter (GH8188)
• SQL io functions now accept a SQLAlchemy connectable. (GH7877)
• pd.read_sql and to_sql can accept database URI as con parameter (GH10214)
• read_sql_table will now allow reading from views (GH10750).
• Enable writing complex values to HDFStores when using the table format (GH10447)
• Enable pd.read_hdf to be used without specifying a key when the HDF file contains a single dataset (GH10443)
• pd.read_stata will now read Stata 118 type files. (GH9882)
• msgpack submodule has been updated to 0.4.6 with backward compatibility (GH10581)
• DataFrame.to_dict now accepts orient='index' keyword argument (GH10844).
• DataFrame.apply will return a Series of dicts if the passed function returns a dict and reduce=True (GH8735).
• Allow passing kwargs to the interpolation methods (GH10378).
• Improved error message when concatenating an empty iterable of DataFrame objects (GH9157)
• pd.read_csv can now read bz2-compressed files incrementally, and the C parser can read bz2-compressed files from AWS S3 (GH11070, GH11072).
• In pd.read_csv, recognize s3n:// and s3a:// URLs as designating S3 file storage (GH11070, GH11071).
Backwards incompatible API changes

Changes to sorting API

The sorting API has had some longtime inconsistencies. (GH9816, GH8239).

Here is a summary of the API PRIOR to 0.17.0:

- `Series.sort` is INPLACE while `DataFrame.sort` returns a new object.
- `Series.order` returns a new object
- It was possible to use `Series/DataFrame.sort_index` to sort by values by passing the by keyword.
- `Series/DataFrame.sortlevel` worked only on a MultiIndex for sorting by index.

To address these issues, we have revamped the API:

- We have introduced a new method, `DataFrame.sort_values()`, which is the merger of `DataFrame.sort()`, `Series.sort()`, and `Series.order()`, to handle sorting of values.
- The existing methods `Series.sort()`, `Series.order()`, and `DataFrame.sort()` have been deprecated and will be removed in a future version.
- The by argument of `DataFrame.sort_index()` has been deprecated and will be removed in a future version.
- The existing method `.sort_index()` will gain the level keyword to enable level sorting.

We now have two distinct and non-overlapping methods of sorting. A * marks items that will show a FutureWarning.

To sort by the values:

<table>
<thead>
<tr>
<th>Previous</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Series.order()</td>
<td>Series.sort_values()</td>
</tr>
<tr>
<td>* Series.sort()</td>
<td>Series.sort_values(inplace=True)</td>
</tr>
<tr>
<td>* DataFrame.sort(columns=...)</td>
<td>DataFrame.sort_values(by=...)</td>
</tr>
</tbody>
</table>

To sort by the index:

<table>
<thead>
<tr>
<th>Previous</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.sort_index()</td>
<td>Series.sort_index()</td>
</tr>
<tr>
<td>Series.sortlevel(level=...)</td>
<td>Series.sort_index(level=...)</td>
</tr>
<tr>
<td>DataFrame.sort_index()</td>
<td>DataFrame.sort_index()</td>
</tr>
<tr>
<td>DataFrame.sortlevel(level=...)</td>
<td>DataFrame.sort_index(level=...)</td>
</tr>
<tr>
<td>* DataFrame.sort()</td>
<td>DataFrame.sort_index()</td>
</tr>
</tbody>
</table>

We have also deprecated and changed similar methods in two Series-like classes, `Index` and `Categorical`.

<table>
<thead>
<tr>
<th>Previous</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Index.order()</td>
<td>Index.sort_values()</td>
</tr>
<tr>
<td>* Categorical.order()</td>
<td>Categorical.sort_values()</td>
</tr>
</tbody>
</table>
Changes to `to_datetime` and `to_timedelta`

**Error handling**

The default for `pd.to_datetime` error handling has changed to `errors='raise'`. In prior versions it was `errors='ignore'`. Furthermore, the `coerce` argument has been deprecated in favor of `errors='coerce'`. This means that invalid parsing will raise rather than return the original input as in previous versions. (GH10636)

Previous behavior:

```python
In [2]: pd.to_datetime(['2009-07-31', 'asd'])
Out[2]: array(['2009-07-31', 'asd'], dtype=object)
```

New behavior:

```python
In [3]: pd.to_datetime(['2009-07-31', 'asd'])
ValueError: Unknown string format
```

Of course you can coerce this as well.

```python
In [61]: pd.to_datetime(['2009-07-31', 'asd'], errors='coerce')
Out[61]: DatetimeIndex(['2009-07-31', NaT'], dtype='datetime64[ns]', freq=None)
```

To keep the previous behavior, you can use `errors='ignore'`:

```python
In [62]: pd.to_datetime(['2009-07-31', 'asd'], errors='ignore')
Out[62]: Index(['2009-07-31', 'asd'], dtype='object')
```

Furthermore, `pd.to_timedelta` has gained a similar API, of `errors='raise' | 'ignore' | 'coerce'`, and the `coerce` keyword has been deprecated in favor of `errors='coerce'`.

**Consistent parsing**

The string parsing of `to_datetime`, `Timestamp` and `DatetimeIndex` has been made consistent. (GH7599)

Prior to 0.17.0, `Timestamp` and `to_datetime` may parse year-only datetime-string incorrectly using today's date, otherwise `DatetimeIndex` uses the beginning of the year. `Timestamp` and `to_datetime` may raise `ValueError` in some types of datetime-string which `DatetimeIndex` can parse, such as a quarterly string.

Previous behavior:

```python
In [1]: pd.Timestamp('2012Q2')
Traceback...
    ... ValueError: Unable to parse 2012Q2

# Results in today's date.
In [2]: pd.Timestamp('2014')
Out [2]: 2014-08-12 00:00:00
```

v0.17.0 can parse them as below. It works on `DatetimeIndex` also.

New behavior:
Note: If you want to perform calculations based on today's date, use `Timestamp.now()` and `pandas.tseries.offsets`.

```python
In [66]: import pandas.tseries.offsets as offsets
```

```python
In [67]: pd.Timestamp.now()
Out[67]: Timestamp('2022-04-02 08:20:33.602289')
```

```python
In [68]: pd.Timestamp.now() + offsets.DateOffset(years=1)
Out[68]: Timestamp('2023-04-02 08:20:33.603038')
```

### Changes to Index comparisons

Operator equal on `Index` should behave similarly to `Series` ([GH9947, GH10637](https://issues.tidyverse.org/)).

Starting in v0.17.0, comparing `Index` objects of different lengths will raise a `ValueError`. This is to be consistent with the behavior of `Series`.

**Previous behavior:**

```python
In [2]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[2]: array([ True, False, False], dtype=bool)
```

```python
In [3]: pd.Index([1, 2, 3]) == pd.Index([2])
Out[3]: array([False, True, False], dtype=bool)
```

```python
In [4]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
Out[4]: False
```

**New behavior:**

```python
In [8]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[8]: array([ True, False, False], dtype=bool)
```

```python
In [9]: pd.Index([1, 2, 3]) == pd.Index([2])
ValueError: Lengths must match to compare
```

```python
In [10]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
ValueError: Lengths must match to compare
```

Note that this is different from the `numpy` behavior where a comparison can be broadcast:
or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

### Changes to boolean comparisons vs. None

Boolean comparisons of a `Series` vs `None` will now be equivalent to comparing with `np.nan`, rather than raise `TypeError` (GH1079).

```
In [71]: s = pd.Series(range(3))
In [72]: s.iloc[1] = None
In [73]: s
Out[73]:
0    0.0
1    NaN
2    2.0
Length: 3, dtype: float64
```

Previous behavior:

```
In [5]: s == None
TypeError: Could not compare <type 'NoneType'> type with Series
```

New behavior:

```
In [74]: s == None
Out[74]:
0   False
1   False
2   False
Length: 3, dtype: bool
```

Usually you simply want to know which values are null.

```
In [75]: s.isnull()
Out[75]:
0   False
1   True
2   False
Length: 3, dtype: bool
```

**Warning:** You generally will want to use `isnull/notnull` for these types of comparisons, as `isnull/notnull` tells you which elements are null. One has to be mindful that `nan`'s don't compare equal, but `None`'s do. Note that pandas/numpy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`. 

5.14. Version 0.17
HDFStore dropna behavior

The default behavior for HDFStore write functions with format='table' is now to keep rows that are all missing. Previously, the behavior was to drop rows that were all missing save the index. The previous behavior can be replicated using the dropna=True option. (GH9382)

Previous behavior:

```python
In [78]: df_with_missing = pd.DataFrame(
    ....:     {"col1": [0, np.nan, 2], "col2": [1, np.nan, np.nan]}
    ....: )
    ....:
In [79]: df_with_missing
Out[79]:
   col1  col2
0   0.0   1.0
1   NaN  NaN
2   2.0   NaN
[3 rows x 2 columns]
```

```python
In [27]:
df_with_missing.to_hdf('file.h5', 'df_with_missing', format='table', mode='w')
```

```python
In [28]:
pd.read_hdf('file.h5', 'df_with_missing')
```

Out [28]:
```
   col1  col2
0   0.0   1.0
2   NaN  NaN
```

New behavior:

```python
In [80]: df_with_missing.to_hdf("file.h5", "df_with_missing", format="table", mode="w")
```

```python
In [81]:
pd.read_hdf("file.h5", "df_with_missing")
```

Out [81]:
```
   col1  col2
0   0.0   1.0
1   NaN  NaN
2   NaN  NaN
```

(continues on next page)
See the docs for more details.

**Changes to display.precision option**

The display.precision option has been clarified to refer to decimal places (GH10451).

Earlier versions of pandas would format floating point numbers to have one less decimal place than the value in display.precision.

```
In [1]: pd.set_option('display.precision', 2)

In [2]: pd.DataFrame({'x': [123.456789]})
Out[2]:
x
0  123.5
```

If interpreting precision as “significant figures” this did work for scientific notation but that same interpretation did not work for values with standard formatting. It was also out of step with how numpy handles formatting.

Going forward the value of display.precision will directly control the number of places after the decimal, for regular formatting as well as scientific notation, similar to how numpy’s precision print option works.

```
In [82]: pd.set_option("display.precision", 2)

In [83]: pd.DataFrame({"x": [123.456789]})
Out[83]:
x
0  123.46
```

To preserve output behavior with prior versions the default value of display.precision has been reduced to 6 from 7.

**Changes to Categorical.unique**

Categorical.unique now returns new Categoricals with categories and codes that are unique, rather than returning np.array (GH10508)

- unordered category: values and categories are sorted by appearance order.
- ordered category: values are sorted by appearance order, categories keep existing order.

```
In [84]: cat = pd.Categorical(['C', 'A', 'B', 'C'], categories=['A', 'B', 'C'], ordered=True)

In [85]: cat
Out[85]:
['C', 'A', 'B', 'C']
```
Changes to bool passed as header in parsers

In earlier versions of pandas, if a bool was passed the `header` argument of `read_csv, read_excel, or read_html` it was implicitly converted to an integer, resulting in `header=0` for `False` and `header=1` for `True` (GH6113)

A bool input to `header` will now raise a `TypeError`

```
In [29]: df = pd.read_csv('data.csv', header=False)
TypeError: Passing a bool to header is invalid. Use header=None for no header or header=int or list-like of ints to specify the row(s) making up the column names
```

Other API changes

- Line and kde plot with `subplots=True` now uses default colors, not all black. Specify `color='k'` to draw all lines in black (GH9894)
- Calling the `.value_counts()` method on a Series with a `categorical` dtype now returns a Series with a `CategoricalIndex` (GH10704)
- The metadata properties of subclasses of pandas objects will now be serialized (GH10553).
- `groupby` using `Categorical` follows the same rule as `Categorical.unique` described above (GH10508)
- When constructing DataFrame with an array of `complex64` dtype previously meant the corresponding column was automatically promoted to the `complex128` dtype. pandas will now preserve the itemsize of the input for complex data (GH10952)
- some numeric reduction operators would return `ValueError`, rather than `TypeError` on object types that includes strings and numbers (GH11131)
- Passing currently unsupported `chunksize` argument to `read_excel` or `ExcelFile.parse` will now raise `NotImplementedError` (GH8011)
- Allow an `ExcelFile` object to be passed into `read_excel` (GH11198)
• DatetimeIndex.union does not infer freq if self and the input have None as freq (GH11086)
• NaT’s methods now either raise ValueError, or return np.nan or NaT (GH9513)

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>return np.nan</td>
<td>weekday, isoweekday</td>
</tr>
<tr>
<td>return NaT</td>
<td>date, now, replace, to_datetime, today</td>
</tr>
<tr>
<td>return np.datetime64('NaT')</td>
<td>to_datetime64 (unchanged)</td>
</tr>
<tr>
<td>raise ValueError</td>
<td>All other public methods (names not beginning with underscores)</td>
</tr>
</tbody>
</table>

Deprecations

• For Series the following indexing functions are deprecated (GH10177).

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>.irow(i)</td>
<td>.iloc[i] or iat[i]</td>
</tr>
<tr>
<td>.iget(i)</td>
<td>.iloc[i] or iat[i]</td>
</tr>
<tr>
<td>.iget_value(i)</td>
<td>.iloc[i] or iat[i]</td>
</tr>
</tbody>
</table>

• For DataFrame the following indexing functions are deprecated (GH10177).

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>.irow(i)</td>
<td>.iloc[i]</td>
</tr>
<tr>
<td>.iget_value(i, j)</td>
<td>.iloc[i, j] or iat[i, j]</td>
</tr>
<tr>
<td>.icol(j)</td>
<td>.iloc[:, j]</td>
</tr>
</tbody>
</table>

Note: These indexing function have been deprecated in the documentation since 0.11.0.

• Categorical.name was deprecated to make Categorical more numpy.ndarray like. Use Series(cat, name="whatever") instead (GH10482).
• Setting missing values (NaN) in Categorical’s categories will issue a warning (GH10748). You can still have missing values in the values.
• drop_duplicates and duplicated’s take_last keyword was deprecated in favor of keep. (GH6511, GH8505)
• Series.nsmallest and nlargest’s take_last keyword was deprecated in favor of keep. (GH10792)
• DataFrame.combineAdd and DataFrame.combineMult are deprecated. They can easily be replaced by using the add and mul methods: DataFrame.add(other, fill_value=0) and DataFrame.mul(other, fill_value=1.) (GH10735).
• TimeSeries deprecated in favor of Series (note that this has been an alias since 0.13.0), (GH10890)
• SparsePanel deprecated and will be removed in a future version (GH11157).
• Series.is_time_series deprecated in favor of Series.index.is_all_dates (GH11135)
• Legacy offsets (like 'A@JAN') are deprecated (note that this has been alias since 0.8.0) (GH10878)
• WidePanel deprecated in favor of Panel, LongPanel in favor of DataFrame (note these have been aliases since < 0.11.0), (GH10892)
- `DataFrame.convert_objects` has been deprecated in favor of type-specific functions `pd.to_datetime`, `pd.to_timestamp` and `pd.to_numeric` (new in 0.17.0) (GH11133).

**Removal of prior version deprecations(changes)**

- Removal of `na_last` parameters from `Series.order()` and `Series.sort()`, in favor of `na_position` (GH5231)
- Removal of `percentile_width` from `.describe()`, in favor of `percentiles` (GH7088)
- Removal of `colSpace` parameter from `DataFrame.to_string()`, in favor of `col_space`, circa 0.8.0 version.
- Removal of automatic time-series broadcasting (GH2304)

```
In [90]: np.random.seed(1234)
```

```
In [91]: df = pd.DataFrame(
....:     np.random.randn(5, 2),
....:     columns=list("AB"),
....:     index=pd.date_range("2013-01-01", periods=5),
....:     )
```

```
In [92]: df
```

```
Out[92]:
     A       B
2013-01-01  0.471435 -1.190976
2013-01-02  1.432707 -0.312652
2013-01-03 -0.720589  0.887163
2013-01-04  0.859588 -0.636524
2013-01-05  0.015696 -2.242685
[5 rows x 2 columns]
```

Previously

```
In [3]: df + df.A
FutureWarning: TimeSeries broadcasting along DataFrame index by default is deprecated.
Please use DataFrame.<op> to explicitly broadcast arithmetic operations along the index
```

```
Out[3]:
     A       B
2013-01-01  0.942870 -0.719541
2013-01-02  2.865414  1.120055
2013-01-03 -1.441177  0.166574
2013-01-04  1.719177  0.223065
2013-01-05  0.031393 -2.226989
```

Current

```
In [93]: df.add(df.A, axis="index")
```

```
Out[93]:
```
(continues on next page)
A B
2013-01-01 0.942870 -0.719541
2013-01-02 2.865414 1.120055
2013-01-03 -1.441177 0.166574
2013-01-04 1.719177 0.223065
2013-01-05 0.031393 -2.226989

[5 rows x 2 columns]

- Remove **table** keyword in HDFStore.put/append, in favor of using **format=** (GH4645)
- Remove kind in read_excel/ExcelFile as its unused (GH4712)
- Remove infer_type keyword from pd.read_html as its unused (GH4770, GH7032)
- Remove **offset** and **timeRule** keywords from Series.tshift/shift, in favor of **freq** (GH4853, GH4864)
- Remove pd.load/pd.save aliases in favor of pd.to_pickle/pd.read_pickle (GH3787)

### Performance improvements

- Development support for benchmarking with the **Air Speed Velocity** library (GH8361)
- Added vbench benchmarks for alternative ExcelWriter engines and reading Excel files (GH7171)
- Performance improvements in Categorical.value_counts (GH10804)
- Performance improvements in SeriesGroupBy.nunique and SeriesGroupBy.value_counts and SeriesGroupBy.transform (GH10820, GH11077)
- Performance improvements in DataFrame.drop_duplicates with integer dtypes (GH10917)
- Performance improvements in DataFrame.duplicated with wide frames. (GH10161, GH11180)
- 4x improvement in timedelta string parsing (GH6755, GH10426)
- 8x improvement in timedelta64 and datetime64 ops (GH6755)
- Significantly improved performance of indexing MultiIndex with slicers (GH10287)
- 8x improvement in iloc using list-like input (GH10791)
- Improved performance of Series.isin for datetimelike/integer Series (GH10827)
- 20x improvement in concat of Categoricals when categories are identical (GH10587)
- Improved performance of to_datetime when specified format string is ISO8601 (GH10178)
- 2x improvement of Series.value_counts for float dtype (GH10821)
- Enable infer_datetime_format in to_datetime when date components do not have 0 padding (GH11142)
- Regression from 0.16.1 in constructing DataFrame from nested dictionary (GH11084)
- Performance improvements in addition/subtraction operations for DateOffset with Series or DatetimeIndex (GH10744, GH11205)
Bug fixes

- Bug in incorrect computation of .mean() on timedelta64[ns] because of overflow (GH9442)
- Bug in .isin on older numpies (GH11232)
- Bug in DataFrame.to_html(index=False) renders unnecessary name row (GH10344)
- Bug in DataFrame.to_latex() the column_format argument could not be passed (GH9402)
- Bug in DatetimeIndex when localizing with NaT (GH10477)
- Bug in Series.dt ops in preserving meta-data (GH10477)
- Bug in preserving NaT when passed in an otherwise invalid to_datetime construction (GH10477)
- Bug in DataFrame.apply when function returns categorical series. (GH9573)
- Bug in to_datetime with invalid dates and formats supplied (GH10154)
- Bug in Index.drop_duplicates dropping name(s) (GH10115)
- Bug in Series.quantile dropping name (GH10881)
- Bug in pd.Series when setting a value on an empty Series whose index has a frequency. (GH10193)
- Bug in pd.Series.interpolate with invalid order keyword values. (GH10633)
- Bug in DataFrame.plot raises ValueError when color name is specified by multiple characters (GH10387)
- Bug in Index construction with a mixed list of tuples (GH10697)
- Bug in DataFrame.reset_index when index contains NaT. (GH10388)
- Bug in ExcelReader when worksheet is empty (GH6403)
- Bug in BinGrouper.group_info where returned values are not compatible with base class (GH10914)
- Bug in clearing the cache on DataFrame.pop and a subsequent inplace op (GH10912)
- Bug in indexing with a mixed-integer Index causing an ImportError (GH10610)
- Bug in Series.count when index has nulls (GH10946)
- Bug in pickling of a non-regular freq DatetimeIndex (GH11002)
- Bug causing DataFrame.where to not respect the axis parameter when the frame has a symmetric shape. (GH9736)
- Bug in Table.select_column where name is not preserved (GH10392)
- Bug in offsets.generate_range where start and end have finer precision than offset (GH9907)
- Bug in pd.rolling_* where Series.name would be lost in the output (GH10565)
- Bug in stack when index or columns are not unique. (GH10417)
- Bug in setting a Panel when an axis has a MultiIndex (GH10360)
- Bug in USFederalHolidayCalendar where USMemorialDay and USMartinLutherKingJr were incorrect (GH10278 and GH9760)
- Bug in .sample() where returned object, if set, gives unnecessary SettingWithCopyWarning (GH10738)
- Bug in .sample() where weights passed as Series were not aligned along axis before being treated positionally, potentially causing problems if weight indices were not aligned with sampled object. (GH10738)
- Regression fixed in (GH9311, GH6620, GH9345), where groupby with a datetime-like converting to float with certain aggregators (GH10979)
• Bug in DataFrame.interpolate with axis=1 and inplace=True (GH10395)
• Bug in io.sql.get_schema when specifying multiple columns as primary key (GH10385).
• Bug in groupby(sort=False) with datetime-like Categorical raises ValueError (GH10505)
• Bug in groupby(axis=1) with filter() throws IndexError (GH11041)
• Bug in test_categorical on big-endian builds (GH10425)
• Bug in Series.shift and DataFrame.shift not supporting categorical data (GH9416)
• Bug in Series.map using categorical Series raises AttributeError (GH10324)
• Bug in MultiIndex.get_level_values including Categorical raises AttributeError (GH10460)
• Bug in pd.get_dummies with sparse=True not returning SparseDataFrame (GH10531)
• Bug in Index subtypes (such as PeriodIndex) not returning their own type for .drop and .insert methods (GH10620)
• Bug in allos.outer_join_indexer when right array is empty (GH10618)
• Bug in filter (regression from 0.16.0) and transform when grouping on multiple keys, one of which is datetime-like (GH10114)
• Bug in to_datetime and to_timedelta causing Index name to be lost (GH10875)
• Bug in len(DataFrame.groupby) causing IndexError when there’s a column containing only NaNs (GH11016)
• Bug that caused segfault when resampling an empty Series (GH10228)
• Bug in DatetimeIndex and PeriodIndex.value_counts resets name from its result, but retains in result's Index. (GH10150)
• Bug in pd.eval using numexpr engine coerces 1 element numpy array to scalar (GH10546)
• Bug in pd.concat with axis=0 when column is of dtype category (GH10177)
• Bug in read_msgpack where input type is not always checked (GH10369, GH10630)
• Bug in pd.read_csv with kwargs index_col=False, index_col=['a', 'b'] or dtype (GH10413, GH10467, GH10577)
• Bug in Series.from_csv with header kwarg not setting the Series.name or the Series.index.name (GH10483)
• Bug in groupby.var which caused variance to be inaccurate for small float values (GH10448)
• Bug in Series.plot(kind='hist') Y Label not informative (GH10485)
• Bug in read_csv when using a converter which generates a uint8 type (GH9266)
• Bug causes memory leak in time-series line and area plot (GH9003)
• Bug when setting a Panel sliced along the major or minor axes when the right-hand side is a DataFrame (GH11014)
• Bug that returns None and does not raise NotImplementedError when operator functions (e.g. .add) of Panel are not implemented (GH7692)
• Bug in line and kde plot cannot accept multiple colors when subplots=True (GH9894)
• Bug in DataFrame.plot raises ValueError when color name is specified by multiple characters (GH10387)
• Bug in left and right join of Series with MultiIndex may be inverted (GH10665)
• Bug in left and right join of with MultiIndex may be inverted (GH10741)
• Bug in read_stata when reading a file with a different order set in columns (GH10757)
• Bug in Categorical may not representing properly when category contains tz or Period (GH10713)
• Bug in Categorical.__iter__ may not returning correct datetime and Period (GH10713)
• Bug in indexing with a PeriodIndex on an object with a PeriodIndex (GH4125)
• Bug in read_csv with engine='c': EOF preceded by a comment, blank line, etc. was not handled correctly (GH10728, GH10548)
• Reading “famafrench” data via DataReader results in HTTP 404 error because of the website url is changed (GH10591).
• Bug in read_msgpack where DataFrame to decode has duplicate column names (GH9618)
• Bug in io.common.get_filepath_or_buffer which caused reading of valid S3 files to fail if the bucket also contained keys for which the user does not have read permission (GH10604)
• Bug in vectorised setting of timestamp columns with python datetime.date and numpy datetime64 (GH10408, GH10412)
• Bug in Index.take may add unnecessary freq attribute (GH10791)
• Bug in merge with empty DataFrame may raise IndexError (GH10824)
• Bug in to_latex where unexpected keyword argument for some documented arguments (GH10888)
• Bug in indexing of large DataFrame where IndexError is uncaught (GH10645 and GH10692)
• Bug in read_csv when using the nrows or chunksize parameters if file contains only a header line (GH9535)
• Bug in serialization of category types in HDF5 in presence of alternate encodings. (GH10366)
• Bug in pd.DataFrame when constructing an empty DataFrame with a string dtype (GH9428)
• Bug in pd.DataFrame.diff when DataFrame is not consolidated (GH10907)
• Bug in pd.unique for arrays with the datetime64 or timedelta64 dtype that meant an array with object dtype was returned instead the original dtype (GH9431)
• Bug in Timedelta raising error when slicing from 0s (GH10583)
• Bug in DatetimeIndex.take and TimedeltaIndex.take may not raise IndexError against invalid index (GH10295)
• Bug in Series([np.nan]).astype('M8[ms]'), which now returns Series([pd.NaT]) (GH10747)
• Bug in PeriodIndex.order reset freq (GH10295)
• Bug in date_range when freq divides end as nanos (GH10885)
• Bug in iloc allowing memory outside bounds of a Series to be accessed with negative integers (GH10779)
• Bug in read_msgpack where encoding is not respected (GH10581)
• Bug preventing access to the first index when using iloc with a list containing the appropriate negative integer (GH10547, GH10779)
• Bug in TimedeltaIndex formatter causing error while trying to save DataFrame with TimedeltaIndex using to_csv (GH10833)
• Bug in DataFrame.where when handling Series slicing (GH10218, GH9558)
• Bug where pd.read_gbq throws ValueError when Bigquery returns zero rows (GH10273)
• Bug in to_json which was causing segmentation fault when serializing 0-rank ndarray (GH9576)
• Bug in plotting functions may raise IndexError when plotted on GridSpec (GH10819)
• Bug in plot result may show unnecessary minor ticklabels (GH10657)
• Bug in groupby incorrect computation for aggregation on DataFrame with NaT (e.g. first, last, min). (GH10590, GH11010)
• Bug when constructing DataFrame where passing a dictionary with only scalar values and specifying columns did not raise an error (GH10856)
• Bug in .var() causing roundoff errors for highly similar values (GH10242)
• Bug in DataFrame.plot(subplots=True) with duplicated columns outputs incorrect result (GH10962)
• Bug in Index arithmetic may result in incorrect class (GH10638)
• Bug in date_range results in empty if freq is negative annually, quarterly and monthly (GH11018)
• Bug in DatetimeIndex cannot infer negative freq (GH11018)
• Remove use of some deprecated numpy comparison operations, mainly in tests. (GH10569)
• Bug in Index dtype may not applied properly (GH11017)
• Bug in io.gbq when testing for minimum google api client version (GH10652)
• Bug in DataFrame construction from nested dict with timedelta keys (GH11129)
• Bug in .fillna against may raise TypeError when data contains datetime dtype (GH7095, GH11153)
• Bug in .groupby when number of keys to group by is same as length of index (GH11185)
• Bug in convert_objects where converted values might not be returned if all null and coerce (GH9589)
• Bug in convert_objects where copy keyword was not respected (GH9589)

Contributors

A total of 112 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Alex Rothberg
• Andrea Bedini +
• Andrew Rosenfeld
• Andy Hayden
• Andy Li +
• Anthonios Partheniou +
• Artemy Kolchinsky
• Bernard Willers
• Charlie Clark +
• Chris +
• Chris Whelan
• Christoph Gohlke +
• Christopher Whelan
• Clark Fitzgerald
• Clearfield Christopher +
• Dan Ringwald +
• Daniel Ni +
• Data & Code Expert Experimenting with Code on Data +
• David Cottrell
• David John Gagne +
• David Kelly +
• ETF +
• Eduardo Schettino +
• Egor +
• Egor Panfilov +
• Evan Wright
• Frank Pinter +
• Gabriel Araujo +
• Garrett-R
• Gianluca Rossi +
• Guillaume Gay
• Guillaume Poulin
• Harsh Nisar +
• Ian Henriksen +
• Ian Hoegen +
• Jaidev Deshpande +
• Jan Rudolph +
• Jan Schulz
• Jason Swails +
• Jeff Reback
• Jonas Buyl +
• Joris Van den Bossche
• Joris Vankerschaver +
• Josh Levy-Kramer +
• Julien Danjou
• Ka Wo Chen
• Karrie Kehoe +
• Kelsey Jordahl
• Kerby Shedden
• Kevin Sheppard
• Lars Buitinck
• Leif Johnson +
• Luis Ortiz +
• Mac +
• Matt Gambogi +
• Matt Savoie +
• Matthew Gilbert +
• Maximilian Roos +
• Michelangelo D’Agostino +
• Mortada Mehyar
• Nick Eubank
• Nipun Batra
• Ondřej Čertík
• Phillip Cloud
• Pratap Vardhan +
• Rafal Skolasinski +
• Richard Lewis +
• Rinoc Johnson +
• Rob Levy
• Robert Gieseke
• Safia Abdalla +
• Samuel Denny +
• Saumitra Shahapure +
• Sebastian Pölsterl +
• Sebastian Rubbert +
• Sheppard, Kevin +
• Sinhrks
• Siu Kwan Lam +
• Skipper Seabold
• Spencer Carrucciu +
• Stephan Hoyer
• Stephen Hoover +
• Stephen Pascoe +
• Terry Santegoeds +
• Thomas Grainger
• Tjerk Santegoeds +
• Tom Augspurger
5.15 Version 0.16

5.15.1 Version 0.16.2 (June 12, 2015)

This is a minor bug-fix release from 0.16.1 and includes a large number of bug fixes along some new features (pipe() method), enhancements, and performance improvements.

We recommend that all users upgrade to this version.

Highlights include:

• A new pipe method, see here
• Documentation on how to use numba with pandas, see here
What's new in v0.16.2

- **New features**
  - **Pipe**
  - **Other enhancements**
- **API changes**
- **Performance improvements**
- **Bug fixes**
- **Contributors**

**New features**

**Pipe**

We've introduced a new method `DataFrame.pipe()`. As suggested by the name, `pipe` should be used to pipe data through a chain of function calls. The goal is to avoid confusing nested function calls like

```python
# df is a DataFrame
# f, g, and h are functions that take and return DataFrames
f(g(h(df), arg1=1), arg2=2, arg3=3)  # noqa F821
```

The logic flows from inside out, and function names are separated from their keyword arguments. This can be rewritten as

```python
(df.pipe(h)  # noqa F821
 .pipe(g, arg1=1)  # noqa F821
 .pipe(f, arg2=2, arg3=3)  # noqa F821
)
```

Now both the code and the logic flow from top to bottom. Keyword arguments are next to their functions. Overall the code is much more readable.

In the example above, the functions `f`, `g`, and `h` each expected the DataFrame as the first positional argument. When the function you wish to apply takes its data anywhere other than the first argument, pass a tuple of `(function, keyword)` indicating where the DataFrame should flow. For example:

```python
In [1]: import statsmodels.formula.api as sm

In [2]: bb = pd.read_csv("data/baseball.csv", index_col="id")

# sm.ols takes (formula, data)
In [3]: ( ...
 ...:      bb.query("h > 0")
 ...:      .assign(ln_h=lambda df: np.log(df.h))
 ...:      .pipe((sm.ols, "data"), "hr ~ ln_h + year + g + C(lg)")
 ...:      .fit()
 ...:      .summary()
```

(continues on next page)
```python
...
...
```

Out[3]:
```
<class 'statsmodels.iolib.summary.Summary'>
```...

```
OLS Regression Results
==============================================================================
Dep. Variable: hr R-squared: 0.685
Model: OLS Adj. R-squared: 0.665
Method: Least Squares F-statistic: 34.28
Date: Sat, 02 Apr 2022 Prob (F-statistic): 3.48e-15
Time: 08:20:33 Log-Likelihood: -205.92
No. Observations: 68 AIC: 421.8
Df Residuals: 63 BIC: 432.9
Df Model: 4
Covariance Type: nonrobust
===============================================================================
              coef       std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept    -8484.7720   4664.146       -1.819    0.074    -1.78e+04    835.780
C(lg)[T.NL]  -2.2736     1.325        -1.716    0.091     -4.922     0.375
ln_h          -1.3542     0.875       -1.547    0.127    -3.103     0.395
year          4.2277     2.324        1.819    0.074     -0.417     8.872
g             0.1841     0.029        6.258    0.000     0.125     0.243
```

```
Omnibus: 10.875 Durbin-Watson: 1.999
Prob(Omnibus): 0.004 Jarque-Bera (JB): 17.298
Skew: 0.537 Prob(JB): 0.000175
Kurtosis: 5.225 Cond. No. 1.49e+07
```

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are strong multicollinearity or other numerical problems.

The pipe method is inspired by unix pipes, which stream text through processes. More recently dplyr and magrittr have introduced the popular (%%) pipe operator for R.

See the documentation for more. (GH10129)
Other enhancements

- Added rsplit to Index/Series StringMethods (GH10303)
- Removed the hard-coded size limits on the DataFrame HTML representation in the IPython notebook, and leave this to IPython itself (only for IPython v3.0 or greater). This eliminates the duplicate scroll bars that appeared in the notebook with large frames (GH10231).

  Note that the notebook has a toggle output scrolling feature to limit the display of very large frames (by clicking left of the output). You can also configure the way DataFrames are displayed using the pandas options, see here.

- axis parameter of DataFrame.quantile now accepts also index and column. (GH9543)

API changes

- Holiday now raises NotImplementedError if both offset and observance are used in the constructor instead of returning an incorrect result (GH10217).

Performance improvements

- Improved Series.resample performance with dtype=datetime64[ns] (GH7754)
- Increase performance of str.split when expand=True (GH10081)

Bug fixes

- Bug in Series.hist raises an error when a one row Series was given (GH10214)
- Bug where HDFStore.select modifies the passed columns list (GH7212)
- Bug in Categorical repr with display.width of None in Python 3 (GH10087)
- Bug in to_json with certain orients and a CategoricalIndex would segfault (GH10317)
- Bug where some of the nan functions do not have consistent return dtypes (GH10251)
- Bug in DataFrame.quantile on checking that a valid axis was passed (GH9543)
- Bug in groupby.apply aggregation for Categorical not preserving categories (GH10138)
- Bug in to_csv where date_format is ignored if the datetime is fractional (GH10209)
- Bug in DataFrame.to_json with mixed data types (GH10289)
- Bug in cache updating when consolidating (GH10264)
- Bug in mean() where integer dtypes can overflow (GH10172)
- Bug where Panel.from_dict does not set dtype when specified (GH10058)
- Bug in Index.union raises AttributeError when passing array-likes. (GH10149)
- Bug in Timestamp's' : microsecond, quarter, dayofyear, week and daysinmonth properties return np.int type, not built-in int. (GH10050)
- Bug in NaT raises AttributeError when accessing to daysinmonth, dayofweek properties. (GH10096)
- Bug in Index repr when using the max_seq_items=None setting (GH10182).
- Bug in getting timezone data with dateutil on various platforms (GH9059, GH8639, GH9663, GH10121)
pandas: powerful Python data analysis toolkit, Release 1.4.2

• Bug in displaying datetimes with mixed frequencies; display ‘ms’ datetimes to the proper precision. (GH10170)
• Bug in setitem where type promotion is applied to the entire block (GH10280)
• Bug in Series arithmetic methods may incorrectly hold names (GH10068)
• Bug in GroupBy.get_group when grouping on multiple keys, one of which is categorical. (GH10132)
• Bug in DatetimeIndex and TimedeltaIndex names are lost after timedelta arithmetic (GH9926)
• Bug in DataFrame construction from nested dict with datetime64 (GH10160)
• Bug in Series.plot(label="LABEL") not correctly setting the label (GH10119)
• Bug in plot not defaulting to matplotlib axes.grid setting (GH9792)
• Bug causing strings containing an exponent, but no decimal to be parsed as int instead of float in engine='python' for the read_csv parser (GH9565)
• Bug in Series.align resets name when fill_value is specified (GH10067)
• Bug in read_csv causing index name not to be set on an empty DataFrame (GH10184)
• Bug in SparseSeries.abs resets name (GH10241)
• Bug in TimedeltaIndex slicing may reset freq (GH10292)
• Bug in GroupBy.get_group raises ValueError when group key contains NaN (GH6992)
• Bug in SparseSeries constructor ignores input data name (GH10258)
• Bug in Categorical.remove_categories causing a ValueError when removing the NaN category if underlying dtype is floating-point (GH10156)
• Bug where infer_freq infers time rule (WOM-5XXX) unsupported by to_offset (GH9425)
• Bug in DataFrame.to_hdf() where table format would raise a seemingly unrelated error for invalid (non-string) column names. This is now explicitly forbidden. (GH9057)
• Bug to handle masking empty DataFrame (GH10126).
• Bug where MySQL interface could not handle numeric table/column names (GH10255)
• Bug in read_csv with a date_parser that returned a datetime64 array of other time resolution than [ns] (GH10245)
• Bug in Panel.apply when the result has ndim=0 (GH10332)
• Bug in read_hdf where auto_close could not be passed (GH9327).
• Bug in read_hdf where open stores could not be used (GH10330).
• Bug in adding empty DataFrames, now results in a DataFrame that .equals an empty DataFrame (GH10181).
• Bug in to_hdf and HDFStore which did not check that complib choices were valid (GH4582, GH8874).
Contributors

A total of 34 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Andrew Rosenfeld
- Artemy Kolchinsky
- Bernard Willers +
- Christer van der Meeren
- Christian Hudon +
- Constantine Glen Evans +
- Daniel Julius Lasiman +
- Evan Wright
- Francesco Brundu +
- Gaëtan de Menten +
- Jake VanderPlas
- James Hiebert +
- Jeff Reback
- Joris Van den Bossche
- Justin Lecher +
- Ka Wo Chen +
- Kevin Sheppard
- Mortada Mehyar
- Morton Fox +
- Robin Wilson +
- Sinhrks
- Stephan Hoyer
- Thomas Grainger
- Tom Ajamian
- Tom Augspurger
- Yoshiki Vázquez Baeza
- Younggun Kim
- austinc +
- behzad nouri
- jreback
- lexical
- rekcahpassyla +
- scls19fr
5.15.2 Version 0.16.1 (May 11, 2015)

This is a minor bug-fix release from 0.16.0 and includes a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

• Support for a `CategoricalIndex`, a category based index, see [here](#)
• New section on how-to-contribute to `pandas`, see [here](#)
• Revised “Merge, join, and concatenate” documentation, including graphical examples to make it easier to understand each operations, see [here](#)
• New method `sample` for drawing random samples from Series, DataFrames and Panels. See [here](#)
• The default `Index` printing has changed to a more uniform format, see [here](#)
• `BusinessHour` datetime-offset is now supported, see [here](#)
• Further enhancement to the `.str` accessor to make string operations easier, see [here](#)

What’s new in v0.16.1

- **Enhancements**
  - `CategoricalIndex`
  - `Sample`
  - `String methods enhancements`
  - `Other enhancements`
- **API changes**
  - `Deprecations`
- **Index representation**
- **Performance improvements**
- **Bug fixes**
- **Contributors**

**Warning:** In pandas 0.17.0, the sub-package `pandas.io.data` will be removed in favor of a separately installable package (GH8961).
Enhancements

CategoricalIndex

We introduce a CategoricalIndex, a new type of index object that is useful for supporting indexing with duplicates. This is a container around a Categorical (introduced in v0.15.0) and allows efficient indexing and storage of an index with a large number of duplicated elements. Prior to 0.16.1, setting the index of a DataFrame/Series with a category dtype would convert this to regular object-based Index.

```
In [1]: df = pd.DataFrame({'A': np.arange(6),
    ...:                     'B': pd.Series(list('aabbca'))
    ...:                     .astype('category', categories=list('cab'))
    ...:                     })
```

```
In [2]: df
Out[2]:
   A  B
  0  0  a
  1  1  a
  2  2  b
  3  3  b
  4  4  c
  5  5  a
```

```
In [3]: df.dtypes
Out[3]:
A    int64
B  category
dtype: object
```

```
In [4]: df.B.cat.categories
Out[4]: Index(['c', 'a', 'b'], dtype='object')
```

setting the index, will create a CategoricalIndex

```
In [5]: df2 = df.set_index('B')
```

```
In [6]: df2.index
Out[6]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'], _ordered=False, name='B', dtype='category')
```

indexing with __getitem__/.iloc/.loc/.ix works similarly to an Index with duplicates. The indexers MUST be in the category or the operation will raise.

```
In [7]: df2.loc['a']
Out[7]:
   A  B
  a  0
  a  1
  a  5
```

and preserves the CategoricalIndex
In [8]: df2.loc['a'].index
Out[8]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')

sorting will order by the order of the categories

In [9]: df2.sort_index()
Out[9]:
   A  B
c  4
a  0
a  1
a  5
b  2
b  3

groupby operations on the index will preserve the index nature as well

In [10]: df2.groupby(level=0).sum()
Out[10]:
   A  B
c  4
a  6
b  5

In [11]: df2.groupby(level=0).sum().index
Out[11]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')

reindexing operations, will return a resulting index based on the type of the passed indexer, meaning that passing a list will return a plain-old-Index; indexing with a Categorical will return a CategoricalIndex, indexed according to the categories of the PASSED Categorical dtype. This allows one to arbitrarily index these even with values NOT in the categories, similarly to how you can reindex ANY pandas index.

In [12]: df2.reindex(['a', 'e'])
Out[12]:
   A  B
a  0.0
a  1.0
a  5.0
e  NaN

In [13]: df2.reindex(['a', 'e']).index
Out[13]: pd.Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [14]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde')))
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [15]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde'))).index
Out[15]: pd.CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c', 'd', 'e'], ordered=False, name='B', dtype='category')

See the documentation for more. (GH7629, GH10038, GH10039)

Sample

Series, DataFrames, and Panels now have a new method: sample(). The method accepts a specific number of rows or columns to return, or a fraction of the total number of rows or columns. It also has options for sampling with or without replacement, for passing in a column for weights for non-uniform sampling, and for setting seed values to facilitate replication. (GH2419)

In [1]: example_series = pd.Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1
In [2]: example_series.sample()
Out[2]:
3
3
Length: 1, dtype: int64

# One may specify either a number of rows:
In [3]: example_series.sample(n=3)
Out[3]:
2
2
1
1
0
0
Length: 3, dtype: int64

# Or a fraction of the rows:
In [4]: example_series.sample(frac=0.5)
Out[4]:
1
1
5
5
3
3
Length: 3, dtype: int64

# weights are accepted.
In [5]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [6]: example_series.sample(n=3, weights=example_weights)
Out[6]:
2
2
4
4
3
3
Length: 3, dtype: int64

# weights will also be normalized if they do not sum to one,
# and missing values will be treated as zeros.
In [7]: example_weights2 = [0.5, 0, 0, None, np.nan]

In [8]: example_series.sample(n=1, weights=example_weights2)
Out[8]:
0
Length: 1, dtype: int64

When applied to a DataFrame, one may pass the name of a column to specify sampling weights when sampling from rows.

In [9]: df = pd.DataFrame({"col1": [9, 8, 7, 6], "weight_column": [0.5, 0.4, 0.1, 0]})

In [10]: df.sample(n=3, weights="weight_column")
Out[10]:
coll  weight_column
0 9 0.5
1 8 0.4
2 7 0.1

[3 rows x 2 columns]

**String methods enhancements**

*Continuing from v0.16.0*, the following enhancements make string operations easier and more consistent with standard Python string operations.

- Added StringMethods (.str accessor) to Index (GH9068)

  The .str accessor is now available for both Series and Index.

  In [11]: idx = pd.Index(["jack", "jill", "jesse", "frank"])

  In [12]: idx.str.strip()
  Out[12]: Index(["jack", "jill", "jesse", "frank"], dtype='object')

One special case for the .str accessor on Index is that if a string method returns bool, the .str accessor will return a np.array instead of a boolean Index (GH8875). This enables the following expression to work naturally:

In [13]: idx = pd.Index(["a1", "a2", "b1", "b2"])

In [14]: s = pd.Series(range(4), index=idx)

In [15]: s
Out[15]:
a1 0
a2 1
b1 2
The following new methods are accessible via .str accessor to apply the function to each values. (GH9766, GH9773, GH10031, GH10045, GH10052)

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>capitalize()</td>
</tr>
<tr>
<td>swapcase()</td>
</tr>
<tr>
<td>normalize()</td>
</tr>
<tr>
<td>partition()</td>
</tr>
<tr>
<td>rpartition()</td>
</tr>
<tr>
<td>index()</td>
</tr>
<tr>
<td>rindex()</td>
</tr>
<tr>
<td>translate()</td>
</tr>
</tbody>
</table>

- split now takes expand keyword to specify whether to expand dimensionality. return_type is deprecated. (GH9847)

In [18]: s = pd.Series(['a,b', 'a,c', 'b,c'])

    # return Series
    In [19]: s.str.split('',)
      Out[19]:
          0  [a, b]
          1  [a, c]
          2  [b, c]
      Length: 3, dtype: object

    # return DataFrame
    In [20]: s.str.split('', expand=True)
      Out[20]:
          0  1
    0  a  b
    1  a  c
    2  b  c

      [3 rows x 2 columns]

In [21]: idx = pd.Index(['a,b', 'a,c', 'b,c'])

    # return Index
    In [22]: idx.str.split('',)
      Out[22]: Index([['a', 'b'], ['a', 'c'], ['b', 'c']], dtype='object')

    # return MultiIndex
    In [23]: idx.str.split('', expand=True)
      Out[23]: MultiIndex([['a', 'b'],
                          ['a', 'c'],
                          ['b', 'c']],
                          columns=['0', '1'])
Improved extract and get_dummies methods for Index.str (GH9980)

Other enhancements

• BusinessHour offset is now supported, which represents business hours starting from 09:00 - 17:00 on BusinessDay by default. See Here for details. (GH7905)

```python
In [24]: pd.Timestamp("2014-08-01 09:00") + pd.tseries.offsets.BusinessHour()
Out[24]: Timestamp('2014-08-01 10:00:00')

In [25]: pd.Timestamp("2014-08-01 07:00") + pd.tseries.offsets.BusinessHour()
Out[25]: Timestamp('2014-08-01 10:00:00')

In [26]: pd.Timestamp("2014-08-01 16:30") + pd.tseries.offsets.BusinessHour()
Out[26]: Timestamp('2014-08-04 09:30:00')
```

• DataFrame.diff now takes an axis parameter that determines the direction of differencing (GH9727)

• Allow clip, clip_lower, and clip_upper to accept array-like arguments as thresholds (This is a regression from 0.11.0). These methods now have an axis parameter which determines how the Series or DataFrame will be aligned with the threshold(s). (GH6966)

• DataFrame.mask() and Series.mask() now support same keywords as where (GH8801)

• drop function can now accept errors keyword to suppress ValueError raised when any of label does not exist in the target data. (GH6736)

```python
In [27]: df = pd.DataFrame(np.random.randn(3, 3), columns=["A", "B", "C")
In [28]: df.drop(["A", "X"], axis=1, errors="ignore")
Out[28]:
    B       C
0  -0.706771 -1.039575
1   0.424972  0.567020
2  -1.087401  0.673690
[3 rows x 2 columns]
```

• Add support for separating years and quarters using dashes, for example 2014-Q1. (GH9688)

• Allow conversion of values with dtype datetime64 or timedelta64 to strings using astype(str) (GH9757)

• get_dummies function now accepts sparse keyword. If set to True, the return DataFrame is sparse, e.g. SparseDataFrame. (GH8823)

• Period now accepts datetime64 as value input. (GH9054)

• Allow timedelta string conversion when leading zero is missing from time definition, ie 0:00:00 vs 00:00:00. (GH9570)

• Allow Panel.shift with axis='items' (GH9890)
• Trying to write an excel file now raises `NotImplementedError` if the DataFrame has a MultiIndex instead of writing a broken Excel file. (GH9794)
• Allow Categorical.add_categories to accept Series or np.array. (GH9927)
• Add/delete str/dt/cat accessors dynamically from __dir__. (GH9910)
• Add normalize as a dt accessor method. (GH10047)
• DataFrame and Series now have _constructor_expanddim property as overridable constructor for one higher dimensionality data. This should be used only when it is really needed, see [here](#).
• `pd.lib.infer_dtype` now returns `'bytes'` in Python 3 where appropriate. (GH10032)

### API changes

• When passing in an ax to `df.plot( ... , ax=ax)`, the `sharex` kwarg will now default to False. The result is that the visibility of xlabels and xticklabels will not anymore be changed. You have to do that by yourself for the right axes in your figure or set `sharex=True` explicitly (but this changes the visible for all axes in the figure, not only the one which is passed in!). If pandas creates the subplots itself (e.g. no passed in `ax` kwarg), then the default is still `sharex=True` and the visibility changes are applied.
• `assign()` now inserts new columns in alphabetical order. Previously the order was arbitrary. (GH9777)
• By default, `read_csv` and `read_table` will now try to infer the compression type based on the file extension. Set `compression=None` to restore the previous behavior (no decompression). (GH9770)

### Deprecations

• Series.str.split's `return_type` keyword was removed in favor of `expand` (GH9847)

### Index representation

The string representation of `Index` and its sub-classes have now been unified. These will show a single-line display if there are few values; a wrapped multi-line display for a lot of values (but less than `display.max_seq_items`; if lots of items (> `display.max_seq_items`) will show a truncated display (the head and tail of the data). The formatting for MultiIndex is unchanged (a multi-line wrapped display). The display width responds to the option `display.max_seq_items`, which is defaulted to 100. (GH6482)

Previous behavior

```python
In [2]: pd.Index(range(4), name='foo')
Out[2]: Int64Index([0, 1, 2, 3], dtype='int64')

In [3]: pd.Index(range(104), name='foo')
Out[3]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                   20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
                   41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
                   63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
                   84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...], dtype='int64')

In [4]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')
Out[4]: <class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-01-04 00:00:00-05:00]
```

(continues on next page)
New behavior

```
In [29]: pd.set_option("display.width", 80)
```

```
In [30]: pd.Index(range(4), name="foo")
Out[30]: RangeIndex(start=0, stop=4, step=1, name='foo')
```

```
In [31]: pd.Index(range(30), name="foo")
Out[31]: RangeIndex(start=0, stop=30, step=1, name='foo')
```

```
In [32]: pd.Index(range(104), name="foo")
Out[32]: RangeIndex(start=0, stop=104, step=1, name='foo')
```

```
In [33]: pd.CategoricalIndex(["a", "bb", "ccc", "dddd"], ordered=True, name="foobar")
Out[33]: CategoricalIndex(["a", "bb", "ccc", "dddd"], ordered=True, dtype='category', name='foobar')
```

```
In [34]: pd.CategoricalIndex(["a", "bb", "ccc", "dddd"] * 10, ordered=True, name="foobar")
Out[34]: CategoricalIndex(["a", "bb", "ccc", "dddd", 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd'], categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, dtype='category', name='foobar')
```

```
In [35]: pd.CategoricalIndex(["a", "bb", "ccc", "dddd"] * 100, ordered=True, name="foobar")
Out[35]: CategoricalIndex(["a", "bb", "ccc", "dddd", 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd'], categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, dtype='category', name='foobar', length=400)
```

```
In [36]: pd.date_range("20130101", periods=4, name="foo", tz="US/Eastern")
Out[36]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00', '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]', name='foo', freq='D')
```
In [37]: pd.date_range("20130101", periods=25, freq="D")

Out[37]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-25'],
dtype='datetime64[ns]', freq='D')

In [38]: pd.date_range("20130101", periods=104, name="foo", tz="US/Eastern")

Out[38]:
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
               '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00',
               '2013-01-05 00:00:00-05:00', '2013-01-06 00:00:00-05:00',
               '2013-01-07 00:00:00-05:00', '2013-01-08 00:00:00-05:00',
               '2013-01-09 00:00:00-05:00', '2013-01-10 00:00:00-05:00',
               ...
               '2013-04-05 00:00:00-04:00', '2013-04-06 00:00:00-04:00',
               '2013-04-07 00:00:00-04:00', '2013-04-08 00:00:00-04:00',
               '2013-04-09 00:00:00-04:00', '2013-04-10 00:00:00-04:00',
               '2013-04-11 00:00:00-04:00', '2013-04-12 00:00:00-04:00',
               '2013-04-13 00:00:00-04:00', '2013-04-14 00:00:00-04:00'],
dtype='datetime64[ns, US/Eastern]', name='foo', length=104, freq='D')

Performance improvements

- Improved csv write performance with mixed dtypes, including datetimes by up to 5x (GH9940)
- Improved csv write performance generally by 2x (GH9940)
- Improved the performance of pd.lib.max_len_string_array by 5-7x (GH10024)

Bug fixes

- Bug where labels did not appear properly in the legend of DataFrame.plot(), passing label= arguments works, and Series indices are no longer mutated. (GH9542)
- Bug in json serialization causing a segfault when a frame had zero length. (GH9805)
- Bug in read_csv where missing trailing delimiters would cause segfault. (GH5664)
- Bug in retaining index name on appending (GH9862)
- Bug in scatter_matrix draws unexpected axis ticklabels (GH5662)
- Fixed bug in StataWriter resulting in changes to input DataFrame upon save (GH9795).
- Bug in transform causing length mismatch when null entries were present and a fast aggregator was being used (GH9697)
- Bug in equals causing false negatives when block order differed (GH9330)
- Bug in grouping with multiple pd.Grouper where one is non-time based (GH10063)
• Bug in `read_sql_table` error when reading postgres table with timezone (GH7139)
• Bug in DataFrame slicing may not retain metadata (GH9776)
• Bug where TimedeltaIndex were not properly serialized in fixed HDFStore (GH9635)
• Bug with TimedeltaIndex constructor ignoring name when given another TimedeltaIndex as data (GH10025).
• Bug in `DataFrameFormatter._get_formatted_index` with not applying `max_colwidth` to the DataFrame index (GH7856)
• Bug in `.loc` with a read-only ndarray data source (GH10043)
• Bug in `groupby.apply()` that would raise if a passed user defined function either returned only `None` (for all input). (GH9685)
• Always use temporary files in `pytables` tests (GH9992)
• Bug in plotting continuously using `secondary_y` may not show legend properly. (GH9610, GH9779)
• Bug in `DataFrame.plot(kind="hist")` results in `TypeError` when `DataFrame` contains non-numeric columns (GH9853)
• Bug where repeated plotting of `DataFrame` with a `DatetimeIndex` may raise `TypeError` (GH9852)
• Bug in `setup.py` that would allow an incompat cython version to build (GH9827)
• Bug in plotting `secondary_y` incorrectly attaches `right_ax` property to secondary axes specifying itself recursively. (GH9861)
• Bug in `Series.quantile` on empty `Series` of type Datetime or Timedelta (GH9675)
• Bug in `where` causing incorrect results when upcasting was required (GH9731)
• Bug in `FloatArrayFormatter` where decision boundary for displaying “small” floats in decimal format is off by one order of magnitude for a given display.precison (GH9764)
• Fixed bug where `DataFrame.plot()` raised an error when both `color` and `style` keywords were passed and there was no color symbol in the style strings (GH9671)
• Not showing a `DeprecationWarning` on combining list-likes with an `Index` (GH10083)
• Bug in `read_csv` and `read_table` when using `skip_rows` parameter if blank lines are present. (GH9832)
• Bug in `read_csv()` interprets `index_col=True` as 1 (GH9798)
• Bug in index equality comparisons using `==` failing on `Index/MultiIndex` type incompatibility (GH9785)
• Bug in which `SparseDataFrame` could not take `nan` as a column name (GH8822)
• Bug in `to_msgpack` and `read_msgpack` zlib and blosc compression support (GH9783)
• Bug `GroupBy.size` doesn't attach index name properly if grouped by `TimeGrouper` (GH9925)
• Bug causing an exception in slice assignments because `length_of_indexer` returns wrong results (GH9995)
• Bug in `csv` parser causing lines with initial white space plus one non-space character to be skipped. (GH9710)
• Bug in C `csv` parser causing spurious NaNs when data started with newline followed by white space. (GH10022)
• Bug causing elements with a null group to spill into the final group when grouping by a `Categorical` (GH9603)
• Bug where `.iloc` and `.loc` behavior is not consistent on empty dataframes (GH9964)
• Bug in invalid attribute access on a `TimedeltaIndex` incorrectly raised `ValueError` instead of `AttributeError` (GH9680)
• Bug in unequal comparisons between categorical data and a scalar, which was not in the categories (e.g. Series(Categorical(list("abc"), ordered=True)) > "d"). This returned False for all elements, but now raises a TypeError. Equality comparisons also now return False for == and True for !=. (GH9848)

• Bug in DataFrame __setitem__ when right hand side is a dictionary (GH9874)
• Bug in where when dtype is datetime64/timedelta64, but dtype of other is not (GH9804)
• Bug in MultiIndex.sortlevel() results in unicode level name breaks (GH9856)
• Bug in which groupby.transform incorrectly enforced output dtypes to match input dtypes. (GH9807)
• Bug in DataFrame constructor when columns parameter is set, and data is an empty list (GH9939)
• Bug in bar plot with log=True raises TypeError if all values are less than 1 (GH9905)
• Bug in horizontal bar plot ignores log=True (GH9905)
• Bug in PyTables queries that did not return proper results using the index (GH8265, GH9676)
• Bug where dividing a dataframe containing values of type Decimal by another Decimal would raise. (GH9787)
• Bug where using DataFrames asfreq would remove the name of the index. (GH9885)
• Bug causing extra index point when resample BM/BQ (GH9756)
• Changed caching in AbstractHolidayCalendar to be at the instance level rather than at the class level as the latter can result in unexpected behaviour. (GH9552)
• Fixed latex output for MultiIndexed dataframes (GH9778)
• Bug causing an exception when setting an empty range using DataFrame.loc (GH9596)
• Bug in hiding ticklabels with subplots and shared axes when adding a new plot to an existing grid of axes (GH9158)
• Bug in transform and filter when grouping on a categorical variable (GH9921)
• Bug in transform when groups are equal in number and dtype to the input index (GH9700)
• Google BigQuery connector now imports dependencies on a per-method basis. (GH9713)
• Updated BigQuery connector to no longer use deprecated oauth2client.tools.run() (GH8327)
• Bug in subclassed DataFrame. It may not return the correct class, when slicing or subsetting it. (GH9632)
• Bug in .median() where non-float null values are not handled correctly (GH10040)
• Bug in Series.fillna() where it raises if a numerically convertible string is given (GH10092)

Contributors

A total of 58 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Alfonso MHC +
• Andy Hayden
• Artemy Kolchinsky
• Chris Gilmer +
• Chris Grinolds +
• Dan Birken
• David BROCHART +

5.15. Version 0.16
• David Hirschfeld +
• David Stephens
• Dr. Leo +
• Evan Wright +
• Frans van Dumné +
• Hatem Nassrat +
• Henning Sperr +
• Hugo Herter +
• Jan Schulz
• Jeff Blackburne +
• Jeff Reback
• Jim Crist +
• Jonas Abernot +
• Joris Van den Bossche
• Kerby Shedden
• Leo Razoumov +
• Manuel Riel +
• Mortada Mehyar
• Nick Burns +
• Nick Eubank +
• Olivier Grisel
• Phillip Cloud
• Pietro Battiston
• Roy Hyunjin Han
• Sam Zhang +
• Scott Sanderson +
• Sinhrks +
• Stephan Hoyer
• Tiago Antao
• Tom Ajamian +
• Tom Augspurger
• Tomaz Berisa +
• Vikram Shirgur +
• Vladimir Filimonov
• William Hogman +
• Yasin A +
5.15.3 Version 0.16.0 (March 22, 2015)

This is a major release from 0.15.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `DataFrame.assign` method, see [here](#)
- `Series.to_coo/from_coo` methods to interact with `scipy.sparse`, see [here](#)
- Backwards incompatible change to `Timedelta` to conform the `.seconds` attribute with `datetime.timedelta`, see [here](#)
- Changes to the `.loc` slicing API to conform with the behavior of `.ix` see [here](#)
- Changes to the default for ordering in the `Categorical` constructor, see [here](#)
- Enhancement to the `.str` accessor to make string operations easier, see [here](#)
- The `pandas.tools.rplot`, `pandas.sandbox.qtpandas` and `pandas.rpy` modules are deprecated. We refer users to external packages like `seaborn`, `pandas-qt` and `rpy2` for similar or equivalent functionality, see [here](#)

Check the API Changes and deprecations before updating.

What's new in v0.16.0

- **New features**
  - `DataFrame assign`
  - Interaction with `scipy.sparse`
  - String methods enhancements
New features

DataFrame assign

Inspired by dplyr’s `mutate` verb, DataFrame has a new `assign()` method. The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the DataFrame. The new values are inserted, and the entire DataFrame (with all original and new columns) is returned.

```python
In [1]: iris = pd.read_csv('data/iris.data')
In [2]: iris.head()
Out[2]:
     SepalLength  SepalWidth  PetalLength  PetalWidth      Name
0          5.1        3.5          1.4        0.2  Iris-setosa
1          4.9        3.0          1.4        0.2  Iris-setosa
2          4.7        3.2          1.3        0.2  Iris-setosa
3          4.6        3.1          1.5        0.2  Iris-setosa
4          5.0        3.6          1.4        0.2  Iris-setosa
[5 rows x 5 columns]
```

```python
In [3]: iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength']).head()
Out[3]:
     SepalLength  SepalWidth  PetalLength  PetalWidth  Name  sepal_ratio
0          5.1        3.5          1.4        0.2  Iris-setosa  0.686275
1          4.9        3.0          1.4        0.2  Iris-setosa  0.612245
2          4.7        3.2          1.3        0.2  Iris-setosa  0.680851
3          4.6        3.1          1.5        0.2  Iris-setosa  0.673913
4          5.0        3.6          1.4        0.2  Iris-setosa  0.720000
[5 rows x 6 columns]
```

Above was an example of inserting a precomputed value. We can also pass in a function to be evaluated.
The power of `assign` comes when used in chains of operations. For example, we can limit the DataFrame to just those with a Sepal Length greater than 5, calculate the ratio, and plot:

```python
In [5]: iris = pd.read_csv('data/iris.data')

In [6]: (iris.query('SepalLength > 5')
    ...:     .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength, PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
    ...:     .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
```

![Scatter plot of Sepal Ratio vs. Petal Ratio](image)

See the documentation for more. (GH9229)

**Interaction with scipy.sparse**

Added `SparseSeries.to_coo()` and `SparseSeries.from_coo()` methods (GH8048) for converting to and from `scipy.sparse.coo_matrix` instances (see here). For example, given a SparseSeries with MultiIndex we can convert to a `scipy.sparse.coo_matrix` by specifying the row and column labels as index levels:

```python
s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])
s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 1), (1, 1, 'b', 0), (1, 1, 'b', 1), (2, 1, 'b', 0), (2, 1, 'b', 1)],
(continues on next page)
# SparseSeries

```python
s

# SparseSeries
ss = s.to_sparse()
ss

A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
    column_levels=['C', 'D'],
    sort_labels=False)

A
A.todense()
rows
columns
```

The `to_coo` method is a convenience method for creating a `SparseSeries` from a `scipy.sparse.coo_matrix`:

```python
from scipy import sparse
A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
    shape=(3, 4))
A
A.todense()
ss = pd.SparseSeries.from_coo(A)
ss
```

## String methods enhancements

- Following new methods are accessible via `.str` accessor to apply the function to each values. This is intended to make it more consistent with standard methods on strings. (GH9282, GH9352, GH9386, GH9387, GH9439)

<table>
<thead>
<tr>
<th>Method</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum()</td>
<td></td>
</tr>
<tr>
<td>isalpha()</td>
<td></td>
</tr>
<tr>
<td>isdigit()</td>
<td></td>
</tr>
<tr>
<td>isspace()</td>
<td></td>
</tr>
<tr>
<td>islower()</td>
<td></td>
</tr>
<tr>
<td>isupper()</td>
<td></td>
</tr>
<tr>
<td>istitle()</td>
<td></td>
</tr>
<tr>
<td>isnumeric()</td>
<td></td>
</tr>
<tr>
<td>isdecimal()</td>
<td></td>
</tr>
<tr>
<td>find()</td>
<td></td>
</tr>
<tr>
<td>rfind()</td>
<td></td>
</tr>
<tr>
<td>ljust()</td>
<td></td>
</tr>
<tr>
<td>rjust()</td>
<td></td>
</tr>
<tr>
<td>zfill()</td>
<td></td>
</tr>
</tbody>
</table>

```python
In [7]: s = pd.Series(['abcd', '3456', 'EFGH'])

In [8]: s.str.isalpha()
Out[8]:
0   True
1   False
2   True
Length: 3, dtype: bool

In [9]: s.str.find('ab')
Out[9]:
0   0
```

(continues on next page)
Series.str.pad() and Series.str.center() now accept fillchar option to specify filling character (GH9352)

```python
In [10]: s = pd.Series(['12', '300', '25'])
In [11]: s.str.pad(5, fillchar='_')
Out[11]:
0   ___12
1   ____300
2   ____25
Length: 3, dtype: object
```

• Added Series.str.slice_replace(), which previously raised NotImplementedError (GH8888)

```python
In [12]: s = pd.Series(['ABCD', 'EFGH', 'IJK'])
In [13]: s.str.slice_replace(1, 3, 'X')
Out[13]:
0   AXD
1   EXH
2     IX
Length: 3, dtype: object

# replaced with empty char
In [14]: s.str.slice_replace(0, 1)
Out[14]:
0     BCD
1     FGH
2       JK
Length: 3, dtype: object
```

Other enhancements

• Reindex now supports method='nearest' for frames or series with a monotonic increasing or decreasing index (GH9258):

```python
In [15]: df = pd.DataFrame({'x': range(5)})
In [16]: df.reindex([0.2, 1.8, 3.5], method='nearest')
Out[16]:
   x
0  0.2
1  1.8
2  3.5
[3 rows x 1 columns]
```

This method is also exposed by the lower level Index.get_indexer and Index.get_loc methods.
• The `read_excel()` function’s `sheetname` argument now accepts a list and None, to get multiple or all sheets respectively. If more than one sheet is specified, a dictionary is returned. (GH9450)

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel(‘path_to_file.xls', sheetname=[‘Sheet1', 3])
```

• Allow Stata files to be read incrementally with an iterator; support for long strings in Stata files. See the docs [here](GH9493).

• Paths beginning with ~ will now be expanded to begin with the user’s home directory (GH9066)

• Added time interval selection in `get_data_yahoo` (GH9071)

• Added `Timestamp.to_datetime64()` to complement `Timedelta.to_timedelta64()` (GH9255)

• `tseries.frequencies.to_offset()` now accepts `Timedelta` as input (GH9064)

• Lag parameter was added to the autocorrelation method of `Series`, defaults to lag-1 autocorrelation (GH9192)

• `Timedelta` will now accept `nanoseconds` keyword in constructor (GH9273)

• SQL code now safely escapes table and column names (GH8986)

• Added auto-complete for `Series.str.<tab>`, `Series.dt.<tab>` and `Series.cat.<tab>` (GH9322)

• `Index.get_indexer` now supports `method='pad'` and `method='backfill'` even for any target array, not just monotonic targets. These methods also work for monotonic decreasing as well as monotonic increasing indexes (GH9258).

• `Index.asof` now works on all index types (GH9258).

• A `verbose` argument has been augmented in `io.read_excel()`, defaults to False. Set to True to print sheet names as they are parsed. (GH9450)

• Added `days_in_month` (compatibility alias `daysinmonth`) property to `Timestamp`, `DatetimeIndex`, `Period`, `PeriodIndex`, and `Series.dt` (GH9572)

• Added decimal option in `to_csv` to provide formatting for non-`.'` decimal separators (GH781)

• Added normalize option for `Timestamp` to normalized to midnight (GH8794)

• Added example for `DataFrame` import to R using HDF5 file and `rhdf5` library. See the [documentation](GH9636) for more.

**Backwards incompatible API changes**

**Changes in timedelta**

In v0.15.0 a new scalar type `Timedelta` was introduced, that is a sub-class of `datetime.timedelta`. Mentioned [here](GH9185, GH9139) was a notice of an API change w.r.t. the `.seconds` accessor. The intent was to provide a user-friendly set of accessors that give the ‘natural’ value for that unit, e.g. if you had a `Timedelta('1 day, 10:11:12')`, then `.seconds` would return 12. However, this is at odds with the definition of `datetime.timedelta`, which defines `.seconds` as \( 10 \times 3600 + 11 \times 60 + 12 \equiv 36672 \).

So in v0.16.0, we are restoring the API to match that of `datetime.timedelta`. Further, the component values are still available through the `.components` accessor. This affects the `.seconds` and `.microseconds` accessors, and removes the `.hours`, `.minutes`, `.milliseconds` accessors. These changes affect `TimedeltaIndex` and the `Series .dt` accessor as well. (GH9185, GH9139)

Previous behavior
In [2]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [3]: t.days
Out[3]: 1

In [4]: t.seconds
Out[4]: 12

In [5]: t.microseconds
Out[5]: 123

New behavior

In [17]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [18]: t.days
Out[18]: 1

In [19]: t.seconds
Out[19]: 36672

In [20]: t.microseconds
Out[20]: 100123

Using .components allows the full component access

In [21]: t.components
Out[21]: Components(days=1, hours=10, minutes=11, seconds=12, milliseconds=100,
˓→ microseconds=123, nanoseconds=0)

In [22]: t.components.seconds
Out[22]: 12

Indexing changes

The behavior of a small sub-set of edge cases for using .loc have changed (GH8613). Furthermore we have improved the content of the error messages that are raised:

• Slicing with .loc where the start and/or stop bound is not found in the index is now allowed; this previously would raise a KeyError. This makes the behavior the same as .ix in this case. This change is only for slicing, not when indexing with a single label.

In [23]: df = pd.DataFrame(np.random.randn(5, 4),
    ....: columns=list('ABCD'),
    ....: index=pd.date_range('20130101', periods=5))

In [24]: df
Out[24]:
      A        B         C        D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236

(continues on next page)
2013-01-03  -0.861849  -2.104569  -0.494929  1.071804
2013-01-04    0.721555  -0.706771  -1.039575  0.271860
2013-01-05  -0.424972   0.567020   0.276232  -1.087401
[5 rows x 4 columns]

In [25]: s = pd.Series(range(5), [−2, −1, 1, 2, 3])  

In [26]: s
Out[26]:
-2 0
-1 1
 1 2
 2 3
 3 4
Length: 5, dtype: int64

Previous behavior

In [4]: df.loc['2013-01-02':'2013-01-10']
KeyError: 'stop bound [2013-01-10] is not in the [index]'

In [6]: s.loc[-10:3]
KeyError: 'start bound [-10] is not the [index]'

New behavior

In [27]: df.loc['2013-01-02':'2013-01-10']
Out[27]:
   A      B      C      D
2013-01-02  1.212112  -0.173215  0.119209  -1.044236
2013-01-03  -0.861849  -2.104569  -0.494929  1.071804
2013-01-04    0.721555   0.567020   0.276232  -1.087401
2013-01-05  -0.424972  1.071804  -1.087401
[4 rows x 4 columns]

In [28]: s.loc[-10:3]
Out[28]:
-2 0
-1 1
 1 2
 2 3
 3 4
Length: 5, dtype: int64

- Allow slicing with float-like values on an integer index for .ix. Previously this was only enabled for .loc:

Previous behavior

In [8]: s.ix[-1.0:2]
TypeError: the slice start value [-1.0] is not a proper indexer for this index type...
˓

Chapter 5. Release notes
New behavior

In [2]: s.ix[-1.0:2]
Out[2]:
   -1  1
   1  2
   2  3
dtype: int64

• Provide a useful exception for indexing with an invalid type for that index when using .loc. For example trying to use .loc on an index of type DatetimeIndex or PeriodIndex or TimedeltaIndex, with an integer (or a float).

Previous behavior

In [4]: df.loc[2:3]
KeyError: 'start bound [2] is not the [index]'

New behavior

In [4]: df.loc[2:3]
TypeError: Cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'> with <type 'int'> keys

Categorical changes

In prior versions, Categoricals that had an unspecified ordering (meaning no ordered keyword was passed) were defaulted as ordered Categoricals. Going forward, the ordered keyword in the Categorical constructor will default to False. Ordering must now be explicit.

Furthermore, previously you could change the ordered attribute of a Categorical by just setting the attribute, e.g. cat. ordered=True; This is now deprecated and you should use cat.as_ordered() or cat.as_unordered(). These will by default return a new object and not modify the existing object. (GH9347, GH9190)

Previous behavior

In [3]: s = pd.Series([0, 1, 2], dtype='category')

In [4]: s
Out[4]:
   0  0
   1  1
   2  2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [5]: s.cat.ordered
Out[5]: True

In [6]: s.cat.ordered = False

In [7]: s
Out[7]:
   0  0
(continues on next page)
New behavior

```
In [29]: s = pd.Series([0, 1, 2], dtype='category')

In [30]: s
Out[30]:
0 0
1 1
2 2
Length: 3, dtype: category
Categories (3, int64): [0, 1, 2]

In [31]: s.cat.ordered
Out[31]: False

In [32]: s = s.cat.as_ordered()

In [33]: s
Out[33]:
0 0
1 1
2 2
Length: 3, dtype: category
Categories (3, int64): [0 < 1 < 2]

In [34]: s.cat.ordered
Out[34]: True

# you can set in the constructor of the Categorical
In [35]: s = pd.Series(pd.Categorical([0, 1, 2], ordered=True))

In [36]: s
Out[36]:
0 0
1 1
2 2
Length: 3, dtype: category
Categories (3, int64): [0 < 1 < 2]

In [37]: s.cat.ordered
Out[37]: True
```

For ease of creation of series of categorical data, we have added the ability to pass keywords when calling `.astype()`. These are passed directly to the constructor.

```
In [54]: s = pd.Series(["a", "b", "c", "a"]).astype('category', ordered=True)
```
In [55]: s
Out[55]:
0  a
1  b
2  c
3  a
dtype: category
Categories (3, object): [a < b < c]

In [56]: s = (pd.Series(['a', 'b', 'c', 'a'])
       ....: .astype('category', categories=list('abcdef'), ordered=False))

In [57]: s
Out[57]:
0  a
1  b
2  c
3  a
dtype: category
Categories (6, object): [a, b, c, d, e, f]

Other API changes

- Index.duplicated now returns np.array(dtype=bool) rather than Index(dtype=object) containing bool values. (GH8875)
- DataFrame.to_json now returns accurate type serialisation for each column for frames of mixed dtype (GH9037)

Previously data was coerced to a common dtype before serialisation, which for example resulted in integers being serialised to floats:

```python
In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out[2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1.0,"1":2.0}}'
```

Now each column is serialised using its correct dtype:

```python
In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out[2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1,"1":2}}'
```

- DatetimeIndex, PeriodIndex and TimedeltaIndex.summary now output the same format. (GH9116)
- TimedeltaIndex.freqstr now output the same string format as DatetimeIndex. (GH9116)
- Bar and horizontal bar plots no longer add a dashed line along the info axis. The prior style can be achieved with matplotlib's axhline or axvline methods (GH9088).
- Series accessors .dt, .cat and .str now raise AttributeError instead of TypeError if the series does not contain the appropriate type of data (GH9617). This follows Python's built-in exception hierarchy more closely and ensures that tests like hasattr(s, 'cat') are consistent on both Python 2 and 3.
- Series now supports bitwise operation for integral types (GH9016). Previously even if the input dtypes were integral, the output dtype was coerced to bool.
In [2]: pd.Series([0, 1, 2, 3], list('abcd')) | pd.Series([4, 4, 4, 4], list('abcd'))
Out[2]:
a  True
b  True
c  True
d  True
dtype: bool

New behavior. If the input dtypes are integral, the output dtype is also integral and the output values are the result of the bitwise operation.

In [2]: pd.Series([0, 1, 2, 3], list('abcd')) | pd.Series([4, 4, 4, 4], list('abcd'))
Out[2]:
a   4
b   5
c   6
d   7
dtype: int64

• During division involving a Series or DataFrame, 0/0 and 0/0 now give np.nan instead of np.inf. (GH9144, GH8445)

Previous behavior

In [2]: p = pd.Series([0, 1])

In [3]: p / 0
Out[3]:
0   inf
1   inf
dtype: float64

In [4]: p // 0
Out[4]:
0   inf
1   inf
dtype: float64

New behavior

In [38]: p = pd.Series([0, 1])

In [39]: p / 0
Out[39]:
0   NaN
1   inf
Length: 2, dtype: float64

In [40]: p // 0
Out[40]:
0   NaN
1   inf

(continues on next page)
Series.values_counts and Series.describe for categorical data will now put NaN entries at the end. (GH9443)

Series.describe for categorical data will now give counts and frequencies of 0, not NaN, for unused categories (GH9443)

Due to a bug fix, looking up a partial string label with DatetimeIndex.asof now includes values that match the string, even if they are after the start of the partial string label (GH9258).

Old behavior:

```python
In [4]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
Out[4]: Timestamp('2000-01-31 00:00:00')
```

Fixed behavior:

```python
In [41]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
Out[41]: Timestamp('2000-02-28 00:00:00')
```

To reproduce the old behavior, simply add more precision to the label (e.g., use 2000-02-01 instead of 2000-02).

**Deprecations**

- The rplot trellis plotting interface is deprecated and will be removed in a future version. We refer to external packages like seaborn for similar but more refined functionality (GH3445). The documentation includes some examples how to convert your existing code from rplot to seaborn here.

- The pandas.sandbox.qtpandas interface is deprecated and will be removed in a future version. We refer users to the external package pandas-qt. (GH9615)

- The pandas.rpy interface is deprecated and will be removed in a future version. Similar functionality can be accessed through the rpy2 project (GH9602)

- Adding DatetimeIndex/PeriodIndex to another DatetimeIndex/PeriodIndex is being deprecated as a set-operation. This will be changed to a TypeError in a future version. .union() should be used for the union set operation. (GH9094)

- Subtracting DatetimeIndex/PeriodIndex from another DatetimeIndex/PeriodIndex is being deprecated as a set-operation. This will be changed to an actual numeric subtraction yielding a TimeDeltaIndex in a future version. .difference() should be used for the differencing set operation. (GH9094)

**Removal of prior version deprecations/changes**

- DataFrame.pivot_table and crosstab's rows and cols keyword arguments were removed in favor of index and columns (GH6581)

- DataFrame.to_excel and DataFrame.to_csv cols keyword argument was removed in favor of columns (GH6581)

- Removed convert_dummies in favor of get_dummies (GH6581)

- Removed value_range in favor of describe (GH6581)
Performance improvements

- Fixed a performance regression for .loc indexing with an array or list-like (GH9126).
- DataFrame.to_json 30x performance improvement for mixed dtype frames. (GH9037)
- Performance improvements in MultiIndex.duplicated by working with labels instead of values (GH9125)
- Improved the speed of nunique by calling unique instead of value_counts (GH9129, GH7771)
- Performance improvement of up to 10x in DataFrame.count and DataFrame.dropna by taking advantage of homogeneous/heterogeneous dtypes appropriately (GH9136)
- Performance improvement of up to 20x in DataFrame.count when using a MultiIndex and the level keyword argument (GH9163)
- Performance and memory usage improvements in merge when key space exceeds int64 bounds (GH9151)
- Performance improvements in multi-key groupby (GH9429)
- Performance improvements in MultiIndex.sortlevel (GH9445)
- Performance and memory usage improvements in DataFrame.duplicated (GH9398)
- Cythonized Period (GH9440)
- Decreased memory usage on to_hdf (GH9648)

Bug fixes

- Changed .to_html to remove leading/trailing spaces in table body (GH4987)
- Fixed issue using read_csv on s3 with Python 3 (GH9452)
- Fixed compatibility issue in DatetimeIndex affecting architectures where numpy.int_ defaults to numpy.int32 (GH8943)
- Bug in Panel indexing with an object-like (GH9140)
- Bug in the returned Series.dt.components index was reset to the default index (GH9247)
- Bug in Categorical.__getitem__/__setitem__ with listlike input getting incorrect results from indexer coercion (GH9469)
- Bug in partial setting with a DatetimeIndex (GH9478)
- Bug in groupby for integer and datetime64 columns when applying an aggregator that caused the value to be changed when the number was sufficiently large (GH9311, GH6620)
- Fixed bug in to_sql when mapping a Timestamp object column (datetime column with timezone info) to the appropriate sqlalchemy type (GH9085).
- Fixed bug in to_sql dtype argument not accepting an instantiated SQLAlchemy type (GH9083).
- Bug in .loc partial setting with a np.datetime64 (GH9516)
- Incorrect dtypes inferred on datetimelike looking Series & on .xs slices (GH9477)
- Items in Categorical.unique() (and s.unique() if s is of dtype category) now appear in the order in which they are originally found, not in sorted order (GH9331). This is now consistent with the behavior for other dtypes in pandas.
- Fixed bug on big endian platforms which produced incorrect results in StataReader (GH8688).
- Bug in MultiIndex.has_duplicates when having many levels causes an indexer overflow (GH9075, GH5873)
• Bug in pivot and unstack where nan values would break index alignment (GH4862, GH7401, GH7403, GH7405, GH7466, GH9497)
• Bug in left join on MultiIndex with sort=True or null values (GH9210).
• Bug in MultiIndex where inserting new keys would fail (GH9250).
• Bug in groupby when key space exceeds int64 bounds (GH9096).
• Bug in unstack with TimedeltaIndex or DatetimeIndex and nulls (GH9491).
• Bug in rank where comparing floats with tolerance will cause inconsistent behaviour (GH8365).
• Fixed character encoding bug in read_stata and StataReader when loading data from a URL (GH9231).
• Bug in adding offsets.Nano to other offsets raises TypeError (GH9284)
• Bug in DatetimeIndex iteration, related to (GH8890), fixed in (GH9100)
• Bugs in resample around DST transitions. This required fixing offset classes so they behave correctly on DST transitions. (GH5172, GH8744, GH8653, GH9173, GH9468).
• Bug in binary operator method (eg .mul()) alignment with integer levels (GH9463).
• Bug in boxplot, scatter and hexbin plot may show an unnecessary warning (GH8877)
• Bug in subplot with layout kw may show unnecessary warning (GH9464)
• Bug in using grouper functions that need passed through arguments (e.g. axis), when using wrapped function (e.g. fillna), (GH9221)
• DataFrame now properly supports simultaneous copy and dtype arguments in constructor (GH9099)
• Bug in read_csv when using skiprows on a file with CR line endings with the c engine. (GH9079)
• isnull now detects NaT in PeriodIndex (GH9129)
• Bug in groupby .nth() with a multiple column groupby (GH8979)
• Bug in DataFrame.where and Series.where coerce numerics to string incorrectly (GH9280)
• Bug in DataFrame.where and Series.where raise ValueError when string list-like is passed. (GH9280)
• Accessing Series.str methods on with non-string values now raises TypeError instead of producing incorrect results (GH9184)
• Bug in DatetimeIndex.__contains__ when index has duplicates and is not monotonic increasing (GH9512)
• Fixed division by zero error for Series.kurt() when all values are equal (GH9197)
• Fixed issue in the xlsxwriter engine where it added a default ‘General’ format to cells if no other format was applied. This prevented other row or column formatting being applied. (GH9167)
• Fixes issue with index_col=False when usecols is also specified in read_csv. (GH9082)
• Bug where wide_to_long would modify the input stub names list (GH9204)
• Bug in to_sql not storing float64 values using double precision. (GH9009)
• SparseSeries and SparsePanel now accept zero argument constructors (same as their non-sparse counterparts) (GH9272).
• Regression in merging Categorical and object dtypes (GH9426)
• Bug in read_csv with buffer overflows with certain malformed input files (GH9205)
• Bug in groupby MultiIndex with missing pair (GH9049, GH9344)
• Fixed bug in Series.groupby where grouping on MultiIndex levels would ignore the sort argument (GH9444)
• Fix bug in DataFrame.Groupby where sort=False is ignored in the case of Categorical columns. (GH8868)
• Fixed bug with reading CSV files from Amazon S3 on python 3 raising a TypeError (GH9452)
• Bug in the Google BigQuery reader where the 'jobComplete' key may be present but False in the query results (GH8728)
• Bug in Series.values_counts with excluding NaN for categorical type Series with dropna=True (GH9443)
• Fixed missing numeric_only option for DataFrame.std/var/sem (GH9201)
• Support constructing Panel or Panel4D with scalar data (GH8285)
• Series text representation disconnected from max_rows/max_columns (GH7508).

• Series number formatting inconsistent when truncated (GH8532).

Previous behavior

In [2]: pd.options.display.max_rows = 10
In [3]: s = pd.Series([1,1,1,1,1,1,1,1,1,1,0.9999,1,1]*10)
In [4]: s
Out[4]:
0   1
1   1
2   1
...  
127  0.9999
128  1.0000
129  1.0000
Length: 130, dtype: float64

New behavior

0   1.0000
1   1.0000
2   1.0000
3   1.0000
4   1.0000
...  
125 1.0000
126 1.0000
127 0.9999
128 1.0000
129 1.0000
dtype: float64

• A Spurious SettingWithCopy Warning was generated when setting a new item in a frame in some cases (GH8730)

The following would previously report a SettingWithCopy Warning.

In [42]: df1 = pd.DataFrame({'x': pd.Series(['a', 'b', 'c']),
   ....:     'y': pd.Series(['d', 'e', 'f'])})
In [43]: df2 = df1[['x']]
In [44]: df2['y'] = ['g', 'h', 'i']

Contributors

A total of 60 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Toth +
- Alan Du +
- Alessandro Amici +
- Artey Kolchinsky
- Ashwini Chaudhary +
- Ben Schiller
- Bill Letson
- Brandon Bradley +
- Chau Hoang +
- Chris Reynolds
- Chris Whelan +
- Christer van der Meeren +
- David Cottrell +
- David Stephens
- Ehsan Azarnasab +
- Garrett-R +
- Guillaume Gay
- Jake Torcasso +
- Jason Sexauer
- Jeff Reback
- John McNamara
- Joris Van den Bossche
- Joschka zur Jacobsmühlen +
- Juarez Bochi +
- Junya Hayashi +
- K.-Michael Aye
- Kerby Shedden +
- Kevin Sheppard
- Kieran O’Mahony
5.16 Version 0.15

5.16.1 Version 0.15.2 (December 12, 2014)

This is a minor release from 0.15.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. A small number of API changes were necessary to fix existing bugs. We recommend that all users upgrade to this version.

- Enhancements
- API Changes
- Performance Improvements
- Bug Fixes

API changes

- Indexing in MultiIndex beyond lex-sort depth is now supported, though a lexically sorted index will have a better performance. (GH2646)

```
In [1]: df = pd.DataFrame({'jim':[0, 0, 1, 1],
                       ...
                       'joe':['x', 'x', 'z', 'y'],
                       ...
                       'jolie':np.random.rand(4)).set_index(['jim', 'joe'])

In [2]: df
Out[2]:
     jolie
     jim  joe
     0  x  0.126970
     x  0.966718
     1  z  0.260476
     y  0.897237

[4 rows x 1 columns]

In [3]: df.index.lexsort_depth
Out[3]: 1

# in prior versions this would raise a KeyError
# will now show a PerformanceWarning
In [4]: df.loc[(1, 'z')]  
Out[4]:
     jolie
     jim  joe
     1  z  0.260476

[1 rows x 1 columns]

# lexically sorting
In [5]: df2 = df.sort_index()

In [6]: df2
```
Bug in unique of Series with category dtype, which returned all categories regardless whether they were “used” or not (see GH8559 for the discussion). Previous behaviour was to return all categories:

```
In [3]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [4]: cat
Out[4]:
[a, b, a]
Categories (3, object): [a < b < c]

In [5]: cat.unique()
Out[5]: array(['a', 'b', 'c'], dtype=object)
```

Now, only the categories that do effectively occur in the array are returned:

```
In [9]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [10]: cat.unique()
Out[10]:
['a', 'b']
Categories (3, object): ['a', 'b', 'c']
```

- Series.all and Series.any now support the level and skipna parameters. Series.all, Series.any, Index.all, and Index.any no longer support the out and keepdims parameters, which existed for compatibility with ndarray. Various index types no longer support the all and any aggregation functions and will now raise TypeError. (GH8302).

- Allow equality comparisons of Series with a categorical dtype and object dtype; previously these would raise TypeError (GH8938)

- Bug in NDFrame: conflicting attribute/column names now behave consistently between getting and setting. Previously, when both a column and attribute named y existed, data.y would return the attribute, while data.y = z would update the column (GH8994)
In [11]: data = pd.DataFrame({'x': [1, 2, 3]})

In [12]: data.y = 2

In [13]: data['y'] = [2, 4, 6]

In [14]: data
Out[14]:
   x  y
0  1  2
1  2  4
2  3  6
[3 rows x 2 columns]

# this assignment was inconsistent
In [15]: data.y = 5

Old behavior:

In [6]: data.y
Out[6]: 2

In [7]: data['y'].values
Out[7]: array([5, 5, 5])

New behavior:

In [16]: data.y
Out[16]: 5

In [17]: data['y'].values
Out[17]: array([2, 4, 6])

• Timestamp('now') is now equivalent to Timestamp.now() in that it returns the local time rather than UTC. Also, Timestamp('today') is now equivalent to Timestamp.today() and both have tz as a possible argument. (GH9000)

• Fix negative step support for label-based slices (GH8753)

Old behavior:

In [1]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])
Out[1]:
   a  0
   b  1
   c  2
dtype: int64

In [2]: s.loc['c':'a':-1]
Out[2]:
   c  2
dtype: int64

New behavior:
Enhancements

Categorical enhancements:

- Added ability to export Categorical data to Stata (GH8633). See [here](#) for limitations of categorical variables exported to Stata data files.
- Added flag `order_categoricals` to `StataReader` and `read_stata` to select whether to order imported categorical data (GH8836). See [here](#) for more information on importing categorical variables from Stata data files.
- Added ability to export Categorical data to/from HDF5 (GH7621). Queries work the same as if it was an object array. However, the category dtyped data is stored in a more efficient manner. See [here](#) for an example and caveats w.r.t. prior versions of pandas.
- Added support for `searchsorted()` on Categorical class (GH8420).

Other enhancements:

- Added the ability to specify the SQL type of columns when writing a DataFrame to a database (GH8778). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```python
from sqlalchemy.types import String
data.to_sql('data_dtype', engine, dtype={'Col_1': String})  # noqa F821
```

- `Series.all` and `Series.any` now support the `level` and `skipna` parameters (GH8302):

```python
In [20]: s = pd.Series([False, True, False], index=[0, 0, 1])
In [21]: s.any(level=0)
Out[21]:
0    True
1    False
Length: 2, dtype: bool
```

- Panel now supports the `all` and `any` aggregation functions. (GH8302):

```python
>>> p = pd.Panel(np.random.rand(2, 5, 4) > 0.1)
>>> p.all()
```

```python
0  1  2  3
0 True True True True
1 True False True True
2 True True True True
3 False True False True
4 True True True True
```

- Added support for `utcfromtimestamp()`, `fromtimestamp()`, and `combine()` on `Timestamp` class (GH5351).
• Added Google Analytics (pandas.io.ga) basic documentation (GH8835). See here.
• Timedelta arithmetic returns NotImplemented in unknown cases, allowing extensions by custom classes (GH8813).
• Timedelta now supports arithmetic with numpy.ndarray objects of the appropriate dtype (numpy 1.8 or newer only) (GH8884).
• Added Timedelta.to_timedelta64() method to the public API (GH8884).
• Added gbq.generate_bq_schema() function to the gbq module (GH8325).
• Series now works with map objects the same way as generators (GH8909).
• Added context manager to HDFStore for automatic closing (GH8791).
• to_datetime gains an exact keyword to allow for a format to not require an exact match for a provided format string (if its False). exact defaults to True (meaning that exact matching is still the default) (GH8904)
• Added axvlines boolean option to parallel_coordinates plot function, determines whether vertical lines will be printed, default is True
• Added ability to read table footers to read_html (GH8552)
• to_sql now infers data types of non-NA values for columns that contain NA values and have dtype object (GH8778).

Performance

• Reduce memory usage when skiprows is an integer in read_csv (GH8681)
• Performance boost for to_datetime conversions with a passed format=, and the exact=False (GH8904)

Bug fixes

• Bug in concat of Series with category dtype which were coercing to object. (GH8641)
• Bug in Timestamp-Timestamp not returning a Timedelta type and datelike-datelike ops with timezones (GH8865)
• Made consistent a timezone mismatch exception (either tz operated with None or incompatible timezone), will now return TypeError rather than ValueError (a couple of edge cases only), (GH8865)
• Bug in using a pd.Grouper(key=...) with no level/axis or level only (GH8795, GH8866)
• Report a TypeError when invalid/no parameters are passed in a groupby (GH8015)
• Bug in packaging pandas with py2app/cx_Freeze (GH8602, GH8831)
• Bug in groupby signatures that didn’t include *args or **kwargs (GH8733).
• io.data.Options now raises RemoteDataError when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
• Unclear error message in csv parsing when passing dtype and names and the parsed data is a different data type (GH8833)
• Bug in slicing a MultiIndex with an empty list and at least one boolean indexer (GH8781)
• io.data.Options now raises RemoteDataError when no expiry dates are available from Yahoo (GH8761).
• Timedelta kwargs may now be numpy ints and floats (GH8757).
• Fixed several outstanding bugs for Timedelta arithmetic and comparisons (GH8813, GH5963, GH5436).
• sql_schema now generates dialect appropriate CREATE TABLE statements (GH8697)
• slice string method now takes step into account (GH8754)
• Bug in BlockManager where setting values with different type would break block integrity (GH8850)
• Bug in DatetimeIndex when using time object as key (GH8667)
• Bug in merge where how='left' and sort=False would not preserve left frame order (GH7331)
• Bug in MultiIndex.reindex where reindexing at level would not reorder labels (GH4088)
• Bug in certain operations with dateutil timezones, manifesting with dateutil 2.3 (GH8639)
• Regression in DatetimeIndex iteration with a Fixed/Local offset timezone (GH8890)
• Bug in to_datetime when parsing a nanoseconds using the %f format (GH8989)
• io.data.Options now raises RemoteDataError when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
• Fix: The font size was only set on x axis if vertical or the y axis if horizontal. (GH8765)
• Fixed division by 0 when reading big csv files in python 3 (GH8621)
• Bug in outputting a MultiIndex with to_html, index=False which would add an extra column (GH8452)
• Imported categorical variables from Stata files retain the ordinal information in the underlying data (GH8836).
• Defined .size attribute across NDFrame objects to provide compat with numpy >= 1.9.1; buggy with np.array_split (GH8846)
• Skip testing of histogram plots for matplotlib <= 1.2 (GH8648).
• Bug where get_data_google returned object dtypes (GH3995)
• Bug in DataFrame.stack(..., dropna=False) when the DataFrame's columns is a MultiIndex whose labels do not reference all its levels. (GH8844)
• Bug in that Option context applied on __enter__ (GH8514)
• Bug in resample that causes a ValueError when resampling across multiple days and the last offset is not calculated from the start of the range (GH8683)
• Bug where DataFrame.plot(kind='scatter') fails when checking if an np.array is in the DataFrame (GH8852)
• Bug in pd.infer_freq/DataFrame.inferred_freq that prevented proper sub-daily frequency inference when the index contained DST days (GH8772).
• Bug where index name was still used when plotting a series with use_index=False (GH8558).
• Bugs when trying to stack multiple columns, when some (or all) of the level names are numbers (GH8584).
• Bug in MultiIndex where __contains__ returns wrong result if index is not lexically sorted or unique (GH7724)
• BUG CSV: fix problem with trailing white space in skipped rows, (GH8679), (GH8661), (GH8983)
• Regression in Timestamp does not parse ‘Z’ zone designator for UTC (GH8771)
• Bug in StataWriter the produces writes strings with 244 characters irrespective of actual size (GH8969)
• Fixed ValueError raised by cummin/cummax when datetime64 Series contains NaT. (GH8965)
• Bug in DataReader returns object dtype if there are missing values (GH8980)
• Bug in plotting if sharex was enabled and index was a time series, would show labels on multiple axes (GH9364).
• Bug where passing a unit to the TimedeltaIndex constructor applied the to nano-second conversion twice. (GH9011).
• Bug in plotting of a period-like array (GH9012)

**Contributors**

A total of 49 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Aaron Staple
• Angelos Evripiotis +
• Artemy Kolchinsky
• Benoit Pointet +
• Brian Jacobowski +
• Charalampos Papaloizou +
• Chris Warth +
• David Stephens
• Fabio Zanini +
• Francesc Via +
• Henry Kleynhans +
• Jake VanderPlas +
• Jan Schulz
• Jeff Reback
• Jeff Tratner
• Joris Van den Bossche
• Kevin Sheppard
• Matt Suggest +
• Matthew Brett
• Phillip Cloud
• Rupert Thompson +
• Scott E Lasley +
• Stephan Hoyer
• Stephen Simmons +
• Sylvain Corlay +
• Thomas Grainger +
• Tiago Antao +
• Tom Augspurger
• Trent Hauck
• Victor Chaves +
• Victor Salgado +
5.16.2 Version 0.15.1 (November 9, 2014)

This is a minor bug-fix release from 0.15.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- Enhancements
- API Changes
- Bug Fixes

API changes

- `s.dt.hour` and other `.dt` accessors will now return `np.nan` for missing values (rather than previously -1), (GH8689)

```python
In [1]: s = pd.Series(pd.date_range("20130101", periods=5, freq="D"))
In [2]: s.iloc[2] = np.nan
In [3]: s
Out[3]:
0    2013-01-01
1    2013-01-02
2    NaT
```

(continues on next page)
previous behavior:

```
In [6]: s.dt.hour
Out[6]:
0   0
1   0
2  -1
3   0
4   0
dtype: int64
```

current behavior:

```
In [4]: s.dt.hour
Out[4]:
0   0.0
1   0.0
2  NaN
3   0.0
4   0.0
Length: 5, dtype: float64
```

- groupby with `as_index=False` will not add erroneous extra columns to result (GH8582):

```
In [5]: np.random.seed(2718281)

In [6]: df = pd.DataFrame(np.random.randint(0, 100, (10, 2)), columns=['jim', 'joe'])

In [7]: df.head()
Out[7]:
   jim  joe
0   61   81
1   96   49
2   55   65
3   72   51
4   77   12
[5 rows x 2 columns]

In [8]: ts = pd.Series(5 * np.random.randint(0, 3, 10))

previous behavior:

```
In [4]: df.groupby(ts, as_index=False).max()
Out[4]:
     NaN  jim  joe
0  0.0  72  83
```

(continues on next page)
current behavior:

```python
In [9]: df.groupby(ts, as_index=False).max()
Out[9]:
      jim  joe
0     72  83
1     77  84
2     96  65
[3 rows x 2 columns]
```

• `groupby` will not erroneously exclude columns if the column name conflicts with the grouper name (GH8112):

```python
In [10]: df = pd.DataFrame({"jim": range(5), "joe": range(5, 10)})
In [11]: df
Out[11]:
     jim  joe
0     0   5
1     1   6
2     2   7
3     3   8
4     4   9
[5 rows x 2 columns]
In [12]: gr = df.groupby(df["jim"] < 2)
```

previous behavior (excludes 1st column from output):

```python
In [4]: gr.apply(sum)
Out[4]:
  joe
jim
False 24
True 11
```

current behavior:

```python
In [13]: gr.apply(sum)
Out[13]:
      jim  joe
    False 9  24
    True  1  11
[2 rows x 2 columns]
```

• Support for slicing with monotonic decreasing indexes, even if `start` or `stop` is not found in the index (GH7860):
In [14]: s = pd.Series(['a', 'b', 'c', 'd'], [4, 3, 2, 1])
In [15]: s
Out[15]:
4   a
3   b
2   c
1   d
Length: 4, dtype: object

Previous behavior:
In [8]: s.loc[3.5:1.5]
KeyError: 3.5

Current behavior:
In [16]: s.loc[3.5:1.5]
Out[16]:
3   b
2   c
Length: 2, dtype: object

- `io.data.Options` has been fixed for a change in the format of the Yahoo Options page (GH8612), (GH8741)

Note: As a result of a change in Yahoo's option page layout, when an expiry date is given, `Options` methods now return data for a single expiry date. Previously, methods returned all data for the selected month.

The `month` and `year` parameters have been undeprecated and can be used to get all options data for a given month.

If an expiry date that is not valid is given, data for the next expiry after the given date is returned.

Option data frames are now saved on the instance as `callsYYMDD` or `putsYYMDD`. Previously they were saved as `callsMMYY` and `putsMMYY`. The next expiry is saved as `calls` and `puts`.

New features:
- The expiry parameter can now be a single date or a list-like object containing dates.
- A new property `expiry_dates` was added, which returns all available expiry dates.

Current behavior:

In [17]: from pandas.io.data import Options
In [18]: aapl = Options('aapl', 'yahoo')
In [19]: aapl.get_call_data().iloc[0:5, 0:1]
Out[19]:
Strike    Expiry    Type Symbol   Last
80  2014-11-14  call AAPL141114C00080000  29.05
84  2014-11-14  call AAPL141114C00084000  24.80
85  2014-11-14  call AAPL141114C00085000  24.05

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

86 2014-11-14 call AAPL141114C00086000 22.76
87 2014-11-14 call AAPL141114C00087000 21.74

In [20]: aapl.expiry_dates
Out[20]:
[datetime.date(2014, 11, 14),
datetime.date(2014, 11, 22),
datetime.date(2014, 11, 28),
datetime.date(2014, 12, 5),
datetime.date(2014, 12, 12),
datetime.date(2014, 12, 20),
datetime.date(2015, 1, 17),
datetime.date(2015, 2, 20),
datetime.date(2015, 4, 17),
datetime.date(2015, 7, 17),
datetime.date(2016, 1, 15),
datetime.date(2017, 1, 20)]

In [21]: aapl.get_near_stock_price(expiry=aapl.expiry_dates[0:3]).iloc[0:5, 0:1]
Out[21]:
<table>
<thead>
<tr>
<th>Strike</th>
<th>Expiry</th>
<th>Type</th>
<th>Symbol</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>2014-11-22 call AAPL141122C00109000</td>
<td>1.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014-11-28 call AAPL141128C00109000</td>
<td>1.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>2014-11-14 call AAPL141114C00110000</td>
<td>0.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014-11-22 call AAPL141122C00110000</td>
<td>1.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014-11-28 call AAPL141128C00110000</td>
<td>1.32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- pandas now also registers the `datetime64` dtype in matplotlib's units registry to plot such values as datetimes. This is activated once pandas is imported. In previous versions, plotting an array of `datetime64` values will have resulted in plotted integer values. To keep the previous behaviour, you can do `del matplotlib.units.registry[np.datetime64]` (GH8614).

Enhancements

- `concat` permits a wider variety of iterables of pandas objects to be passed as the first parameter (GH8645):

  In [17]: from collections import deque
  In [18]: df1 = pd.DataFrame([1, 2, 3])
  In [19]: df2 = pd.DataFrame([4, 5, 6])

  previous behavior:

  In [7]: pd.concat(deque((df1, df2)))
  TypeError: first argument must be a list-like of pandas objects, you passed an object of type "deque"

  current behavior:
In [20]: pd.concat(deque((df1, df2)))
Out[20]:
     0  1  2
   0  3  4
   1  5  6
   2  7  8

[6 rows x 1 columns]

- Represent MultiIndex labels with a dtype that utilizes memory based on the level size. In prior versions, the memory usage was a constant 8 bytes per element in each level. In addition, in prior versions, the reported memory usage was incorrect as it didn’t show the usage for the memory occupied by the underlying data array. (GH8456)

In [21]: dfi = pd.DataFrame(
   ....:       1, index=pd.MultiIndex.from_product(["a", range(1000)], columns=["A"
   ....:       )
   ....:

previous behavior:

# this was underreported in prior versions
In [1]: dfi.memory_usage(index=True)
Out[1]:
Index 8000 # took about 24008 bytes in < 0.15.1
A 8000
dtype: int64

current behavior:

In [22]: dfi.memory_usage(index=True)
Out[22]:
Index 44212
A 8000
Length: 2, dtype: int64

- Added Index properties is_monotonic_increasing and is_monotonic_decreasing (GH8680).
- Added option to select columns when importing Stata files (GH7935)
- Qualify memory usage in DataFrame.info() by adding + if it is a lower bound (GH8578)
- Raise errors in certain aggregation cases where an argument such as numeric_only is not handled (GH8592).
- Added support for 3-character ISO and non-standard country codes in io.wb.download() (GH8482)
- World Bank data requests now will warn/raise based on an errors argument, as well as a list of hard-coded country codes and the World Bank’s JSON response. In prior versions, the error messages didn’t look at the World Bank’s JSON response. Problem-inducing input were simply dropped prior to the request. The issue was that many good countries were cropped in the hard-coded approach. All countries will work now, but some bad countries will raise exceptions because some edge cases break the entire response. (GH8482)
- Added option to Series.str.split() to return a DataFrame rather than a Series (GH8428)
• Added option to `df.info(null_counts=None|True|False)` to override the default display options and force showing of the null-counts (GH8701)

**Bug fixes**

• Bug in unpickling of a `CustomBusinessDay` object (GH8591)
• Bug in coercing `Categorical` to a records array, e.g. `df.to_records()` (GH8626)
• Bug in `Categorical` not created properly with `Series.to_frame()` (GH8626)
• Bug in coercing in `astype` of a `Categorical` of a passed `pd.Categorical` (this now raises `TypeError` correctly), (GH8626)
• Bug in `cut/qcut` when using `Series` and `retbins=True` (GH8589)
• Bug in writing `Categorical` columns to an SQL database with `to_sql` (GH8624).
• Bug in comparing `Categorical` of datetime raising when being compared to a scalar datetime (GH8687)
• Bug in selecting from a `Categorical` with `.iloc` (GH8623)
• Bug in `groupby-transform` with a `Categorical` (GH8623)
• Bug in `duplicated/drop_duplicates` with a `Categorical` (GH8623)
• Bug in `Categorical` reflected comparison operator raising if the first argument was a numpy array scalar (e.g. np.int64) (GH8658)
• Bug in `Panel` indexing with a list-like (GH8710)
• Compat issue is `DataFrame.dtypes` when `options.mode.use_inf_as_null` is True (GH8722)
• Bug in `read_csv`, `dialect` parameter would not take a string (GH8703)
• Bug in slicing a MultiIndex level with an empty-list (GH8737)
• Bug in numeric index operations of add/sub with Float/Index Index with numpy arrays (GH8608)
• Bug in `setitem` with empty indexer and unwanted coercion of dtypes (GH8669)
• Bug in `ix/loc` block splitting on `setitem` (manifests with integer-like dtypes, e.g. datetime64) (GH8607)
• Bug when doing label based indexing with integers not found in the index for non-unique but monotonic indexes (GH8680).
• Bug when indexing a Float64Index with `np.nan` on numpy 1.7 (GH8980).
• Fix `shape` attribute for `MultiIndex` (GH8609)
• Bug in `GroupBy` where a name conflict between the grouper and columns would break `groupby` operations (GH7115, GH8112)
  • Fixed a bug where plotting a column `y` and specifying a label would mutate the index name of the original DataFrame (GH8494)
• Fix regression in plotting of a DatetimeIndex directly with `matplotlib` (GH8614).
• Bug in `date_range` where partially-specified dates would incorporate current date (GH6961)
• Bug in setting by indexer to a scalar value with a mixed-dtype `Panel4d` was failing (GH8702)
• Bug where `DataReader`'s would fail if one of the symbols passed was invalid. Now returns data for valid symbols and `np.nan` for invalid (GH8494)
• Bug in `get_quote_yahoo` that wouldn’t allow non-float return values (GH5229).
Contributors

A total of 23 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Staple +
- Andrew Rosenfeld
- Anton I. Sipos
- Arteemy Kolchinsky
- Bill Letson +
- Dave Hughes +
- David Stephens
- Guillaume Horel +
- Jeff Reback
- Joris Van den Bossche
- Kevin Sheppard
- Nick Stahl +
- Sanghee Kim +
- Stephan Hoyer
- Tom Augspurger
- TomAugspurger
- WANG Aiyong +
- behzad nouri
- immerrr
- jnmclarty
- jreback
- pallav-fdsi +
- unutbu

5.16.3 Version 0.15.0 (October 18, 2014)

This is a major release from 0.14.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Warning: pandas >= 0.15.0 will no longer support compatibility with NumPy versions < 1.7.0. If you want to use the latest versions of pandas, please upgrade to NumPy >= 1.7.0 (GH7711)

- Highlights include:
  - The Categorical type was integrated as a first-class pandas type, see here
  - New scalar type Timedelta, and a new index type TimedeltaIndex, see here
pandas: powerful Python data analysis toolkit, Release 1.4.2

- New datetimelike properties accessor `.dt` for Series, see Datetimelike Properties
- New DataFrame default display for `df.info()` to include memory usage, see Memory Usage
- `read_csv` will now by default ignore blank lines when parsing, see here
- API change in using Indexes in set operations, see here
- Enhancements in the handling of timezones, see here
- A lot of improvements to the rolling and expanding moment functions, see here
- Internal refactoring of the `Index` class to no longer sub-class `ndarray`, see Internal Refactoring
- dropping support for PyTables less than version 3.0.0, and numexpr less than version 2.1 (GH7990)
- Split indexing documentation into Indexing and Selecting Data and MultiIndex / Advanced Indexing
- Split out string methods documentation into Working with Text Data

- Check the API Changes and deprecations before updating
- Other Enhancements
- Performance Improvements
- Bug Fixes

**Warning:** In 0.15.0 `Index` has internally been refactored to no longer sub-class `ndarray` but instead subclass `PandasObject`, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (See the Internal Refactoring)

**Warning:** The refactoring in `Categorical` changed the two argument constructor from “codes/labels and levels” to “values and levels (now called ‘categories’)”. This can lead to subtle bugs. If you use `Categorical` directly, please audit your code before updating to this pandas version and change it to use the `from_codes()` constructor. See more on Categorical here

New features

**Categoricals in Series/DataFrame**

`Categorical` can now be included in `Series` and `DataFrames` and gained new methods to manipulate. Thanks to Jan Schulz for much of this API/implementation. (GH3943, GH5313, GH5314, GH7444, GH7839, GH7848, GH7864, GH7914, GH7768, GH8006, GH3678, GH8075, GH8076, GH8143, GH8453, GH8518).

For full docs, see the categorical introduction and the API documentation.

```
In [1]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
                        ...
                        "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})

In [2]: df["grade"] = df["raw_grade"].astype("category")

In [3]: df["grade"]
Out[3]:
```
# Rename the categories
In [4]: df["grade"].cat.categories = ["very good", "good", "very bad"]

# Reorder the categories and simultaneously add the missing categories
In [5]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [6]: df["grade"]
Out[6]:
0  very good
1   good
2   good
3  very good
4  very good
5   very bad
Name: grade, Length: 6, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']

In [7]: df.sort_values("grade")
Out[7]:
   id  raw_grade  grade
0  5       e    very bad
1  1       b     good
2  3       b     good
3  0       a  very good
4  4       a  very good
5  5       a  very good
[6 rows x 3 columns]

In [8]: df.groupby("grade").size()
Out[8]:
          grade
very bad     1
   bad       0
   medium     0
   good       2
very good     3
Length: 5, dtype: int64

- pandas.core.group_agg and pandas.core.factor_agg were removed. As an alternative, construct a dataframe and use df.groupby(<group>).agg(<func>).
• Supplying “codes/labels and levels” to the Categorical constructor is not supported anymore. Supplying two arguments to the constructor is now interpreted as “values and levels (now called ‘categories’)”. Please change your code to use the from_codes() constructor.

• The Categorical.labels attribute was renamed to Categorical.codes and is read only. If you want to manipulate codes, please use one of the API methods on Categoricals.

• The Categorical.levels attribute is renamed to Categorical.categories.

**TimedeltaIndex/scalar**

We introduce a new scalar type Timedelta, which is a subclass of datetime.timedelta, and behaves in a similar manner, but allows compatibility with np.timedelta64 types as well as a host of custom representation, parsing, and attributes. This type is very similar to how Timestamp works for datetimes. It is a nice-API box for the type. See the docs. (GH3009, GH4533, GH8209, GH8187, GH8190, GH7869, GH7661, GH8345, GH8471)

Warning: Timedelta scalars (and TimedeltaIndex) component fields are not the same as the component fields on a datetime.timedelta object. For example, .seconds on a datetime.timedelta object returns the total number of seconds combined between hours, minutes and seconds. In contrast, the pandas Timedelta breaks out hours, minutes, microseconds and nanoseconds separately.

```python
# Timedelta accessor
In [9]: tds = pd.Timedelta('31 days 5 min 3 sec')

In [10]: tds.minutes
Out[10]: 5L

In [11]: tds.seconds
Out[11]: 3L

# datetime.timedelta accessor
# this is 5 minutes * 60 + 3 seconds
In [12]: tds.to_pytimedelta().seconds
Out[12]: 303

Note: this is no longer true starting from v0.16.0, where full compatibility with datetime.timedelta is introduced. See the 0.16.0 whatsnew entry
```

Warning: Prior to 0.15.0 pd.to_timedelta would return a Series for list-like/Series input, and a np.timedelta64 for scalar input. It will now return a TimedeltaIndex for list-like input, Series for Series input, and Timedelta for scalar input.

The arguments to pd.to_timedelta are now (arg,unit='ns',box=True,coerce=False), previously were (arg,box=True,unit='ns') as these are more logical.

Construct a scalar

```python
In [9]: pd.Timedelta('1 days 06:05:01.000030')
Out[9]: Timedelta('1 days 06:05:01.000030')

In [10]: pd.Timedelta('15.5us')
```

(continues on next page)
Out[10]: Timedelta('0 days 00:00:00.000015500')

In [11]: pd.Timedelta('1 hour 15.5us')
Out[11]: Timedelta('0 days 01:00:00.000015500')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [12]: pd.Timedelta('-1us')
Out[12]: Timedelta('-1 days +23:59:59.999999')

# a NaT
In [13]: pd.Timedelta('nan')
Out[13]: NaT

Access fields for a Timedelta

In [14]: td = pd.Timedelta('1 hour 3m 15.5us')
In [15]: td.seconds
Out[15]: 3780
In [16]: td.microseconds
Out[16]: 15
In [17]: td.nanoseconds
Out[17]: 500

Construct a TimedeltaIndex

In [18]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05',
                          '2 days 00:00:02'],
                          freq=None)

Constructing a TimedeltaIndex with a regular range

In [19]: pd.timedelta_range('1 days', periods=5, freq='D')
Out[19]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'],
                          dtype='timedelta64[ns]', freq='D')

In [20]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')
Out[20]: TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                          '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                          '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                          '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                          '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                          '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                          '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                          ....])
You can now use a `TimedeltaIndex` as the index of a pandas object

```python
In [21]: s = pd.Series(np.arange(5),
   ....:     index=pd.timedelta_range('1 days', periods=5, freq='s'))
   ....:

In [22]: s
Out[22]:
1 days 00:00:00  0
1 days 00:00:01  1
1 days 00:00:02  2
1 days 00:00:03  3
1 days 00:00:04  4
Freq: S, Length: 5, dtype: int64
```

You can select with partial string selections

```python
In [23]: s['1 day 00:00:02']
Out[23]: 2

In [24]: s['1 day':'1 day 00:00:02']
Out[24]:
1 days 00:00:00  0
1 days 00:00:01  1
1 days 00:00:02  2
Freq: S, Length: 3, dtype: int64
```

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are `NaT` preserving:

```python
In [25]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [26]: tdi.tolist()
Out[26]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [27]: dti = pd.date_range('20130101', periods=3)

In [28]: dti.tolist()
Out[28]: [Timestamp('2013-01-01 00:00:00', freq='D'),
Timestamp('2013-01-02 00:00:00', freq='D'),
Timestamp('2013-01-03 00:00:00', freq='D')]
```
In [29]: (dti + tdi).tolist()
Out[29]: [Timestamp('2013-01-05 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [30]: (dti - tdi).tolist()
Out[30]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]

• iteration of a Series e.g. list(Series(...)) of timedelta64[ns] would prior to v0.15.0 return np. timedelta64 for each element. These will now be wrapped in Timedelta.

Memory usage

Implemented methods to find memory usage of a DataFrame. See the FAQ for more. (GH6852).

A new display option display.memory_usage (see Options and settings) sets the default behavior of the memory_usage argument in the df.info() method. By default display.memory_usage is True.

In [31]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
            'complex128', 'object', 'bool']

In [32]: n = 5000
In [33]: data = {t: np.random.randint(100, size=n).astype(t) for t in dtypes}
In [34]: df = pd.DataFrame(data)
In [35]: df['categorical'] = df['object'].astype('category')

In [36]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
# Column      Non-Null Count   Dtype
--- ------ -------------- ----- 
 0 int64    5000 non-null    int64 
 1 float64  5000 non-null    float64 
 2 datetime64[ns]  5000 non-null  datetime64[ns] 
 3 timedelta64[ns]  5000 non-null  timedelta64[ns] 
 4 complex128  5000 non-null   complex128 
 5 object    5000 non-null    object 
 6 bool      5000 non-null    bool 
 7 categorical  5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
        object(1), timedelta64[ns](1)
memory usage: 288.2+ KB

Additionally memory_usage() is an available method for a dataframe object which returns the memory usage of each column.
Series.dt accessor

Series has gained an accessor to succinctly return datetime like properties for the values of the Series, if its a date-time/period like Series.  (GH7207) This will return a Series, indexed like the existing Series.  See the docs

```python
# datetime
In [38]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))
In [39]: s
dt
Out[39]:
0  2013-01-01 09:10:12
1  2013-01-02 09:10:12
2  2013-01-03 09:10:12
3  2013-01-04 09:10:12
Length: 4, dtype: datetime64[ns]

In [40]: s.dt.hour
dt
Out[40]:
0   9
1   9
2   9
3   9
Length: 4, dtype: int64

In [41]: s.dt.second
dt
Out[41]:
0  12
1  12
2  12
3  12
Length: 4, dtype: int64

In [42]: s.dt.day
dt
Out[42]:
0   1
1   2
2   3
3   4
(continues on next page)
This enables nice expressions like this:

```
In [44]: s[s.dt.day == 2]
Out[44]:
1  2013-01-02 09:10:12
Length: 1, dtype: datetime64[ns]
```

You can easily produce tz aware transformations:

```
In [45]: stz = s.dt.tz_localize('US/Eastern')

In [46]: stz
Out[46]:
0  2013-01-01 09:10:12-05:00  
1  2013-01-02 09:10:12-05:00  
2  2013-01-03 09:10:12-05:00  
3  2013-01-04 09:10:12-05:00  
Length: 4, dtype: datetime64[ns, US/Eastern]
```

You can also chain these types of operations:

```
In [48]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[48]:
0  2013-01-01 04:10:12-05:00  
1  2013-01-02 04:10:12-05:00  
2  2013-01-03 04:10:12-05:00  
3  2013-01-04 04:10:12-05:00  
Length: 4, dtype: datetime64[ns, US/Eastern]
```

The .dt accessor works for period and timedelta dtypes.

```
# period
In [49]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [50]: s
dt
Out[50]:
0  2013-01-01  
1  2013-01-02  
2  2013-01-03  
3  2013-01-04  
Length: 4, dtype: period[D]

In [51]: s.dt.year
dt
Out[51]:
0  2013
```
In [52]: s.dt.day
Out[52]:
0 1
1 2
2 3
3 4
Length: 4, dtype: int64

# timedelta
In [53]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [54]: s
Out[54]:
0 1 days 00:00:05
1 1 days 00:00:06
2 1 days 00:00:07
3 1 days 00:00:08
Length: 4, dtype: timedelta64[ns]

In [55]: s.dt.days
Out[55]:
0 1
1 1
2 1
3 1
Length: 4, dtype: int64

In [56]: s.dt.seconds
Out[56]:
0 5
1 6
2 7
3 8
Length: 4, dtype: int64

In [57]: s.dt.components
Out[57]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0   1      0         0        5            0           0          0
1   1      0         0        6            0           0          0
2   1      0         0        7            0           0          0
3   1      0         0        8            0           0          0
[4 rows x 7 columns]
Timezone handling improvements

- `tz_localize(None)` for tz-aware Timestamp and DatetimeIndex now removes timezone holding local time, previously this resulted in Exception or TypeError (GH7812)

```
In [58]: ts = pd.Timestamp('2014-08-01 09:00', tz='US/Eastern')
Out[58]: Timestamp('2014-08-01 09:00:00-0400', tz='US/Eastern')

In [59]: ts
Out[59]: Timestamp('2014-08-01 09:00:00-0400', tz='US/Eastern')

In [60]: ts.tz_localize(None)
Out[60]: Timestamp('2014-08-01 09:00:00')

In [61]: didx = pd.date_range(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')

In [62]: didx
Out[62]: DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
                     '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
                     '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
                     '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
                     '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
                        dtype='datetime64[ns, US/Eastern]', freq='H')

In [63]: didx.tz_localize(None)
Out[63]: DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
                     '2014-08-01 11:00:00', '2014-08-01 12:00:00',
                     '2014-08-01 13:00:00', '2014-08-01 14:00:00',
                     '2014-08-01 15:00:00', '2014-08-01 16:00:00',
                     '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
                        dtype='datetime64[ns]', freq=None)
```

- `tz_localize` now accepts the ambiguous keyword which allows for passing an array of bools indicating whether the date belongs in DST or not, ‘NaT’ for setting transition times to NaT, ‘infer’ for inferring DST/non-DST, and ‘raise’ (default) for an AmbiguousTimeError to be raised. See the docs for more details (GH7943)

- DataFrame.tz_localize and DataFrame.tz_convert now accepts an optional level argument for localizing a specific level of a MultiIndex (GH7846)

- Timestamp.tz_localize and Timestamp.tz_convert now raise TypeError in error cases, rather than Exception (GH8025)

- A timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone (rather than being a naive datetime64[ns]) as object dtype (GH8411)

- Timestamp.__repr__ displays dateutil.tz.tzoffset info (GH7907)
Rolling/expanding moments improvements

• rolling_min(), rolling_max(), rolling_cov(), and rolling_corr() now return objects with all NaN when \( \text{len(arg)} < \text{min_periods} \leq \text{window} \) rather than raising. (This makes all rolling functions consistent in this behavior). (GH7766)

Prior to 0.15.0

```python
In [64]: s = pd.Series([10, 11, 12, 13])
In [15]: pd.rolling_min(s, window=10, min_periods=5)
ValueError: min_periods (5) must be <= window (4)
```

New behavior

```python
In [4]: pd.rolling_min(s, window=10, min_periods=5)
Out[4]:
0     NaN
1     NaN
2     NaN
3     NaN
dtype: float64
```

• rolling_max(), rolling_min(), rolling_sum(), rolling_mean(), rolling_median(), rolling_std(), rolling_var(), rolling_skew(), rolling_kurt(), rolling_quantile(), rolling_cov(), rolling_corr(), rolling_corr_pairwise(), rolling_window(), and rolling_apply() with center=True previously would return a result of the same structure as the input arg with NaN in the final \((\text{window}-1)/2\) entries. Now the final \((\text{window}-1)/2\) entries of the result are calculated as if the input arg were followed by \((\text{window}-1)/2\) NaN values (or with shrinking windows, in the case of rolling_apply()). (GH7925, GH8269)

Prior behavior (note final value is NaN):

```python
In [7]: pd.rolling_sum(Series(range(4)), window=3, min_periods=0, center=True)
Out[7]:
0    1
1    3
2    6
3   NaN
dtype: float64
```

New behavior (note final value is \(5 = \text{sum([2, 3, NaN])}\)):

```python
In [7]: pd.rolling_sum(pd.Series(range(4)), window=3,
                  ....:      min_periods=0, center=True)
Out[7]:
0    1
1    3
2    6
3    5
dtype: float64
```

• rolling_window() now normalizes the weights properly in rolling mean mode (\(\text{mean}=\text{True}\)) so that the calculated weighted means (e.g. ‘triang’, ‘gaussian’) are distributed about the same means as those calculated without weighting (i.e. ‘boxcar’). See the note on normalization for further details. (GH7618)
In [65]: s = pd.Series([10.5, 8.8, 11.4, 9.7, 9.3])

Behavior prior to 0.15.0:

In [39]: pd.rolling_window(s, window=3, win_type='triang', center=True)
Out[39]:
   0    NaN
   1    6.583333
   2    6.883333
   3    6.683333
   4    NaN
dtype: float64

New behavior:

In [10]: pd.rolling_window(s, window=3, win_type='triang', center=True)
Out[10]:
   0    NaN
   1    9.875
   2   10.325
   3   10.025
   4    NaN
dtype: float64

• Removed center argument from all expanding_ functions (see list), as the results produced when center=True did not make much sense. (GH7925)
• Added optional ddof argument to expanding_cov() and rolling_cov(). The default value of 1 is backwards-compatible. (GH8279)
• Documented the ddof argument to expanding_var(), expanding_std(), rolling_var(), and rolling_std(). These functions’ support of a ddof argument (with a default value of 1) was previously undocumented. (GH8064)
• ewma(), ewmstd(), ewmvol(), ewmvar(), ewmcov(), and ewmcorr() now interpret min_periods in the same manner that the rolling_*() and expanding_*() functions do: a given result entry will be NaN if the (expanding, in this case) window does not contain at least min_periods values. The previous behavior was to set to NaN the min_periods entries starting with the first non-NaN value. (GH7977)

Prior behavior (note values start at index 2, which is min_periods after index 0 (the index of the first non-empty value)):

In [66]: s = pd.Series([1, None, None, None, 2, 3])

In [51]: ewma(s, com=3., min_periods=2)
Out[51]:
   0    NaN
   1    NaN
   2   1.000000
   3   1.000000
   4   1.571429
   5  2.189189

dtype: float64

New behavior (note values start at index 4, the location of the 2nd (since min_periods=2) non-empty value):
In [2]: pd.ewma(s, com=3., min_periods=2)
Out[2]:
0   NaN
1   NaN
2   NaN
3   NaN
4  1.759644
5  2.383784
dtype: float64

• `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now have an optional `adjust` argument, just like `ewma()` does, affecting how the weights are calculated. The default value of `adjust` is `True`, which is backwards-compatible. See `Exponentially weighted moment functions` for details. (GH7911)

• `ewma()`, `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now have an optional `ignore_na` argument. When `ignore_na=False` (the default), missing values are taken into account in the weights calculation. When `ignore_na=True` (which reproduces the pre-0.15.0 behavior), missing values are ignored in the weights calculation. (GH7543)

In [7]: pd.ewma(pd.Series([None, 1., 8.]), com=2.)
Out[7]:
0   NaN
1    1.0
2    5.2
dtype: float64

In [8]: pd.ewma(pd.Series([1., None, 8.]), com=2.,
   ....: ignore_na=True) # pre-0.15.0 behavior
Out[8]:
0    1.0
1    1.0
2    5.2
dtype: float64

In [9]: pd.ewma(pd.Series([1., None, 8.]), com=2.,
   ....: ignore_na=False) # new default
Out[9]:
0  1.000000
1  1.000000
2  5.846154
dtype: float64

Warning: By default (`ignore_na=False`) the `ewm*()` functions’ weights calculation in the presence of missing values is different than in pre-0.15.0 versions. To reproduce the pre-0.15.0 calculation of weights in the presence of missing values one must specify explicitly `ignore_na=True`.

• Bug in `expanding_cov()`, `expanding_corr()`, `rolling_cov()`, `rolling_cor()`, `ewmcov()`, and `ewmcorr()` returning results with columns sorted by name and producing an error for non-unique columns; now handles non-unique columns and returns columns in original order (except for the case of two DataFrames with `pairwise=False`, where behavior is unchanged) (GH7542)

• Bug in `rolling_count()` and `expanding_*()` functions unnecessarily producing error message for zero-length data (GH8056)
• Bug in `rolling_apply()` and `expanding_apply()` interpreting `min_periods=0` as `min_periods=1` (GH8080)

• Bug in `expanding_std()` and `expanding_var()` for a single value producing a confusing error message (GH7900)

• Bug in `rolling_std()` and `rolling_var()` for a single value producing NaN (GH7900)

• Bug in `ewmstd()`, `ewmvol()`, `ewmvar()`, and `ewmcov()` calculation of de-biasing factors when `bias=False` (the default). Previously an incorrect constant factor was used, based on `adjust=True`, `ignore_na=True`, and an infinite number of observations. Now a different factor is used for each entry, based on the actual weights (analogous to the usual $N/(N-1)$ factor). In particular, for a single point a value of NaN is returned when `bias=False`, whereas previously a value of (approximately) 0 was returned.

For example, consider the following pre-0.15.0 results for `ewmvar(..., bias=False)`, and the corresponding debiasing factors:

```
In [67]: s = pd.Series([1., 2., 0., 4.])

In [89]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
Out[89]:
          0        1        2
0  1.000000  1.000000  1.000000
1  1.000000  1.000000  1.000000
2  1.000000  1.000000  1.000000
3  1.000000  1.000000  1.000000

In [14]: pd.ewmvar(s, com=2., bias=False)
Out[14]:
         0        1        2
0  NaN      NaN      NaN
1  0.500000  1.210526  4.089069
2  1.210526  4.089069  12.474888
3  4.089069  12.474888  38.389569

In [15]: pd.ewmvar(s, com=2., bias=False) / pd.ewmvar(s, com=2., bias=True)
Out[15]:
         0        1        2
0  NaN      NaN      NaN
1  2.083333  1.583333  1.425439
2  1.583333  1.285714  1.109677
3  1.425439  1.109677  0.983607
```

Note that entry 0 is approximately 0, and the debiasing factors are a constant 1.25. By comparison, the following 0.15.0 results have a NaN for entry 0, and the debiasing factors are decreasing (towards 1.25):

```
In [67]: s = pd.Series([1., 2., 0., 4.])

In [89]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
Out[89]:
          0        1        2
0  1.000000  1.000000  1.000000
1  1.000000  1.000000  1.000000
2  1.000000  1.000000  1.000000
3  1.000000  1.000000  1.000000

In [14]: pd.ewmvar(s, com=2., bias=False)
Out[14]:
         0        1        2
0  NaN      NaN      NaN
1  0.500000  1.210526  4.089069
2  1.210526  4.089069  12.474888
3  4.089069  12.474888  38.389569

In [15]: pd.ewmvar(s, com=2., bias=False) / pd.ewmvar(s, com=2., bias=True)
Out[15]:
         0        1        2
0  NaN      NaN      NaN
1  2.083333  1.583333  1.425439
2  1.583333  1.285714  1.109677
3  1.425439  1.109677  0.983607
```

See *Exponentially weighted moment functions* for details. (GH7912)
Improvements in the SQL IO module

- Added support for a chunksize parameter to `to_sql` function. This allows DataFrame to be written in chunks and avoid packet-size overflow errors (GH8062).
- Added support for a chunksize parameter to `read_sql` function. Specifying this argument will return an iterator through chunks of the query result (GH2908).
- Added support for writing `datetime.date` and `datetime.time` object columns with `to_sql` (GH6932).
- Added support for specifying a schema to read from/write to with `read_sql_table` and `to_sql` (GH7441, GH7952). For example:

  ```python
  df.to_sql('table', engine, schema='other_schema')  # noqa F821
  pd.read_sql_table('table', engine, schema='other_schema')  # noqa F821
  ```

- Added support for writing NaN values with `to_sql` (GH2754).
- Added support for writing datetime64 columns with `to_sql` for all database flavors (GH7103).

Backwards incompatible API changes

Breaking changes

API changes related to `Categorical` (see here for more details):

- The `Categorical` constructor with two arguments changed from “codes/labels and levels” to “values and levels (now called ‘categories’)”. This can lead to subtle bugs. If you use `Categorical` directly, please audit your code by changing it to use the `from_codes()` constructor.

  An old function call like (prior to 0.15.0):

  ```python
  pd.Categorical([0,1,0,2,1], levels=['a', 'b', 'c'])
  ```

  will have to adapted to the following to keep the same behaviour:

  ```python
  In [2]: pd.Categorical.from_codes([0,1,0,2,1], categories=['a', 'b', 'c'])
  Out[2]:
  [a, b, a, c, b]
  Categories (3, object): [a, b, c]
  ```

API changes related to the introduction of the `Timedelta` scalar (see above for more details):

- Prior to 0.15.0 `to_timedelta()` would return a `Series` for list-like/Series input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for Series input, and `Timedelta` for scalar input.

For API changes related to the rolling and expanding functions, see detailed overview above.

Other notable API changes:

- Consistency when indexing with `.loc` and a list-like indexer when no values are found.
In prior versions there was a difference in these two constructs:

- \texttt{df.loc[[3]]} would return a frame reindexed by 3 (with all \texttt{np.nan} values)
- \texttt{df.loc[[3],:]} would raise \texttt{KeyError}.

Both will now raise a \texttt{KeyError}. The rule is that \textit{at least 1} indexer must be found when using a list-like and \texttt{.loc} (GH7999)

Furthermore in prior versions these were also different:

- \texttt{df.loc[[1,3]]} would return a frame reindexed by [1,3]
- \texttt{df.loc[[1,3],:]} would raise \texttt{KeyError}.

Both will now return a frame reindex by [1,3]. E.g.

\begin{verbatim}
In [3]: df.loc[[1, 3]]
Out[3]:
   0 1 a
   3 NaN

In [4]: df.loc[[1, 3], :]
Out[4]:
   0 1 a
   3 NaN
\end{verbatim}

This can also be seen in multi-axis indexing with a Panel.

\begin{verbatim}
>>> p = pd.Panel(np.arange(2 * 3 * 4).reshape(2, 3, 4),
...               items=['ItemA', 'ItemB'],
...               major_axis=[1, 2, 3],
...               minor_axis=['A', 'B', 'C', 'D'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemB
Major_axis axis: 1 to 3
Minor_axis axis: A to D
\end{verbatim}

The following would raise \texttt{KeyError} prior to 0.15.0:

\begin{verbatim}
In [5]:
Out[5]:
   ItemA  ItemD
1  3    NaN
2  7    NaN
3  11   NaN
\end{verbatim}

Furthermore, \texttt{.loc} will raise If no values are found in a MultiIndex with a list-like indexer:
In [70]: s = pd.Series(np.arange(3, dtype='int64'),
       index=pd.MultiIndex.from_product([['A'], ['foo', 'bar', 'baz']],
                             names=['one', 'two'])).sort_index()

In [71]: s
Out[71]:
one two
A  bar  1
   baz  2
  foo  0
Length: 3, dtype: int64

In [72]: try:
   ....: s.loc[['D']]
   ....: except KeyError as e:
   ....:     print("KeyError: " + str(e))
KeyError: "['D'] not in index"

• Assigning values to None now considers the dtype when choosing an ‘empty’ value (GH7941).

Previously, assigning to None in numeric containers changed the dtype to object (or errored, depending on the call). It now uses NaN:

In [73]: s = pd.Series([1, 2, 3])

In [74]: s.loc[0] = None

In [75]: s
Out[75]:
   0  NaN
   1  2.0
   2  3.0
Length: 3, dtype: float64

NaT is now used similarly for datetime containers.

For object containers, we now preserve None values (previously these were converted to NaN values).

In [76]: s = pd.Series(['a', 'b', 'c'])

In [77]: s.loc[0] = None

In [78]: s
Out[78]:
   0  None
   1  b
   2  c
Length: 3, dtype: object

To insert a NaN, you must explicitly use np.nan. See the docs.
• In prior versions, updating a pandas object inplace would not reflect in other python references to this object. (GH8511, GH5104)

    In [79]: s = pd.Series([1, 2, 3])
    In [80]: s2 = s
    In [81]: s += 1.5

    Behavior prior to v0.15.0

    # the original object
    In [5]: s
    Out[5]:
    0   2.5
    1   3.5
    2   4.5
dtype: float64

    # a reference to the original object
    In [7]: s2
    Out[7]:
    0   1
    1   2
    2   3
dtype: int64

    This is now the correct behavior

    # the original object
    In [82]: s
    Out[82]:
    0   2.5
    1   3.5
    2   4.5
    Length: 3, dtype: float64

    # a reference to the original object
    In [83]: s2
    Out[83]:
    0   2.5
    1   3.5
    2   4.5
    Length: 3, dtype: float64

• Made both the C-based and Python engines for read_csv and read_table ignore empty lines in input as well as white space-filled lines, as long as sep is not white space. This is an API change that can be controlled by the keyword parameter skip_blank_lines. See the docs (GH4466)

• A timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone and inserted as object dtype rather than being converted to a naive datetime64[ns] (GH8411).

• Bug in passing a DatetimeIndex with a timezone that was not being retained in DataFrame construction from a dict (GH7822)
In prior versions this would drop the timezone, now it retains the timezone, but gives a column of `object` dtype:

```python
In [84]: i = pd.date_range('1/1/2011', periods=3, freq='10s', tz='US/Eastern')
In [85]: i
Out[85]:
DatetimeIndex(['2011-01-01 00:00:00-05:00', '2011-01-01 00:00:10-05:00',
              '2011-01-01 00:00:20-05:00'],
      dtype='datetime64[ns, US/Eastern]', freq='10S')
In [86]: df = pd.DataFrame({'a': i})
In [87]: df
Out[87]:
   a
0 2011-01-01 00:00:00-05:00
1 2011-01-01 00:00:10-05:00
2 2011-01-01 00:00:20-05:00
[3 rows x 1 columns]
In [88]: df.dtypes
Out[88]:
   a  datetime64[ns, US/Eastern]
Length: 1, dtype: object
```

Previously this would have yielded a column of `datetime64` dtype, but without timezone info.

The behaviour of assigning a column to an existing dataframe as `df['a'] = i` remains unchanged (this already returned an object column with a timezone).

- When passing multiple levels to `stack()`, it will now raise a `ValueError` when the levels aren't all level names or all level numbers (GH7660). See Reshaping by stacking and unstacking.
- Raise a `ValueError` in `df.to_hdf` with 'fixed' format, if `df` has non-unique columns as the resulting file will be broken (GH7761)
- `SettingWithCopy` raise/warnings (according to the option `mode.chained_assignment`) will now be issued when setting a value on a sliced mixed-dtype DataFrame using chained-assignment. (GH7845, GH7950)

```python
In [1]: df = pd.DataFrame(np.arange(0, 9), columns=['count'])
In [2]: df['group'] = 'b'
In [3]: df.iloc[0:5]['group'] = 'a'
/usr/local/bin/ipython:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/#indexing-view-versus-copy
```

- `merge`, `DataFrame.merge`, and `ordered_merge` now return the same type as the `left` argument (GH7737).
- Previously an enlargement with a mixed-dtype frame would act unlike `.append` which will preserve dtypes (related GH2578, GH8176):
In [89]: df = pd.DataFrame([[True, 1], [False, 2]],
                      columns=["female", "fitness"])

In [90]: df
Out[90]:

    female  fitness
  0  True     1
  1 False     2

[2 rows x 2 columns]

In [91]: df.dtypes
Out[91]:

    female  fitness
dtype: bool      int64

# dtypes are now preserved

In [93]: df
Out[93]:

    female  fitness
  0  True     1
  1 False     2
  2 False     2

[3 rows x 2 columns]

In [94]: df.dtypes
Out[94]:

    female  fitness
dtype: bool      int64

• Series.to_csv() now returns a string when path=None, matching the behaviour of DataFrame.to_csv() (GH8215).
• read_hdf now raises IOError when a file that doesn’t exist is passed in. Previously, a new, empty file was created, and a KeyError raised (GH7715).
• DataFrame.info() now ends its output with a newline character (GH8114)
• Concatenating no objects will now raise a ValueError rather than a bare Exception.
• Merge errors will now be sub-classes of ValueError rather than raw Exception (GH8501)
• DataFrame.plot and Series.plot keywords are now have consistent orders (GH8037)
Internal refactoring

In 0.15.0 `Index` has internally been refactored to no longer sub-class `ndarray` but instead subclass `PandasObject`, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (GH5080, GH7439, GH7796, GH8024, GH8367, GH7997, GH8522):

- you may need to unpickle pandas version < 0.15.0 pickles using `pd.read_pickle` rather than `pickle.load`. See pickle docs
- when plotting with a `PeriodIndex`, the matplotlib internal axes will now be arrays of `Period` rather than a `PeriodIndex` (this is similar to how a `DatetimeIndex` passes arrays of `datetimes` now)
- MultiIndexes will now raise similarly to other pandas objects w.r.t. truth testing, see here (GH7897).
- When plotting a `DatetimeIndex` directly with matplotlib’s `plot` function, the axis labels will no longer be formatted as dates but as integers (the internal representation of a `datetime64`). UPDATE This is fixed in 0.15.1, see here.

Deprecations

- The attributes `Categorical labels` and `levels` attributes are deprecated and renamed to `codes` and `categories`.
- The `outtype` argument to `pd.DataFrame.to_dict` has been deprecated in favor of `orient` (GH7840)
- The `convert_dummies` method has been deprecated in favor of `get_dummies` (GH8140)
- The `infer_dst` argument in `tz_localize` will be deprecated in favor of `ambiguous` to allow for more flexibility in dealing with DST transitions. Replace `infer_dst=True` with `ambiguous='infer'` for the same behavior (GH7943). See the docs for more details.
- The top-level `pd.value_range` has been deprecated and can be replaced by `.describe()` (GH8481)
- The `Index` set operations `+` and `-` were deprecated in order to provide these for numeric type operations on certain index types. `+` can be replaced by `.union()` or `|`, and `-` by `.difference()`. Further the method name `Index.diff()` is deprecated and can be replaced by `Index.difference()` (GH8226)

```
# +
pd.Index(['a', 'b', 'c']) + pd.Index(['b', 'c', 'd'])
# should be replaced by
pd.Index(['a', 'b', 'c']).union(pd.Index(['b', 'c', 'd']))
```

```
# -
pd.Index(['a', 'b', 'c']) - pd.Index(['b', 'c', 'd'])
# should be replaced by
pd.Index(['a', 'b', 'c']).difference(pd.Index(['b', 'c', 'd']))
```

- The `infer_types` argument to `read_html()` now has no effect and is deprecated (GH7762, GH7032).
Removal of prior version deprecations/changes

- Remove DataFrame.delevel method in favor of DataFrame.reset_index

Enhancements

Enhancements in the importing/exporting of Stata files:

- Added support for bool, uint8, uint16 and uint32 data types in to_stata (GH7097, GH7365)
- Added conversion option when importing Stata files (GH8527)
- DataFrame.to_stata and StataWriter check string length for compatibility with limitations imposed in dta files where fixed-width strings must contain 244 or fewer characters. Attempting to write Stata dta files with strings longer than 244 characters raises a ValueError. (GH7858)
- read_stata and StataReader can import missing data information into a DataFrame by setting the argument convert_missing to True. When using this options, missing values are returned as StataMissingValue objects and columns containing missing values have object data type. (GH8045)

Enhancements in the plotting functions:

- Added layout keyword to DataFrame.plot. You can pass a tuple of (rows, columns), one of which can be -1 to automatically infer (GH6667, GH8071).
- Allow to pass multiple axes to DataFrame.plot.hist and boxplot (GH5353, GH6970, GH7069)
- Added support for c, colormap and colorbar arguments for DataFrame.plot with kind='scatter' (GH7780)
- Histogram from DataFrame.plot with kind='hist' (GH7809), See the docs.
- Boxplot from DataFrame.plot with kind='box' (GH7998), See the docs.

Other:

- read_csv now has a keyword parameter float_precision which specifies which floating-point converter the C engine should use during parsing, see here (GH8002, GH8044)
- Added searchsorted method to Series objects (GH7447)
- describe() on mixed-types DataFrames is more flexible. Type-based column filtering is now possible via the include/exclude arguments. See the docs (GH8164).

```
In [95]: df = pd.DataFrame({'catA': ['foo', 'foo', 'bar'] * 8,
                        'catB': ['a', 'b', 'c', 'd'] * 6,
                        'numC': np.arange(24),
                        'numD': np.arange(24.) + .5})

In [96]: df.describe(include=['object'])
Out[96]:
          catA  catB
count   24   24
unique   2    4
top     foo   a
freq    16    6

[4 rows x 2 columns]
```
In [97]: df.describe(include=['number', 'object'], exclude=['float'])
   
Out[97]:
      catA  catB  numC
   count 24    24  24.000000
   unique 2    4       NaN
   top   foo  a       NaN
   freq 16    6       NaN
   mean NaN  NaN  11.500000
   std  NaN  NaN   7.071068
   min  NaN  NaN       0.000000
   25% NaN  NaN       5.750000
   50% NaN  NaN       11.500000
   75% NaN  NaN      17.250000
   max NaN  NaN      23.000000

[11 rows x 3 columns]

Requesting all columns is possible with the shorthand ‘all’

In [98]: df.describe(include='all')

Out[98]:
      catA  catB  numC  numD
   count 24    24  24.000000  24.000000
   unique 2    4       NaN     NaN
   top   foo  a       NaN     NaN
   freq 16    6       NaN     NaN
   mean NaN  NaN  11.500000  12.000000
   std  NaN  NaN   7.071068  7.071068
   min  NaN  NaN       0.000000  0.500000
   25% NaN  NaN       5.750000  6.250000
   50% NaN  NaN       11.500000 12.000000
   75% NaN  NaN      17.250000 17.750000
   max NaN  NaN      23.000000 23.500000

[11 rows x 4 columns]

Without those arguments, describe will behave as before, including only numerical columns or, if none are, only categorical columns. See also the docs

• Added split as an option to the orient argument in pd.DataFrame.to_dict. (GH7840)

• The get_dummies method can now be used on DataFrames. By default only categorical columns are encoded as 0’s and 1’s, while other columns are left untouched.

In [99]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
                      'C': [1, 2, 3]})

   
In [100]: pd.get_dummies(df)

Out[100]:
      C A_a A_b B_b B_c
   0     1   1    0    0   1

(continues on next page)
• PeriodIndex supports resolution as the same as DatetimeIndex (GH7708)
• pandas.tseries.holiday has added support for additional holidays and ways to observe holidays (GH7070)
• pandas.tseries.holiday.Holiday now supports a list of offsets in Python3 (GH7070)
• pandas.tseries.holiday.Holiday now supports a days_of_week parameter (GH7070)
• GroupBy.nth() now supports selecting multiple nth values (GH7910)

```
In [101]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [102]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [103]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[103]:
   a   b
2014  1  1
      1
      1
      1
      1
      1
      1
      1
      1
      1
[9 rows x 2 columns]
```

• Period and PeriodIndex supports addition/subtraction with timedelta-likes (GH7966)

If Period freq is D, H, T, L, U, N, Timedelta-like can be added if the result can have same freq. Otherwise, only the same offsets can be added.

```
In [104]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [105]: idx + pd.offsets.Hour(2)
Out[105]:
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00'],
dtype='period[H]')
```

(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [107]: idx + pd.Timedelta('120m')
Out[107]:
PeriodIndex(
    ['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
     '2014-07-01 14:00', '2014-07-01 15:00'],
dtype='period[H]')

In [108]: idx = pd.period_range('2014-07', periods=5, freq='M')
In [109]: idx
Out[109]:
PeriodIndex(
    dtype='period[M]')

In [110]: idx + pd.offsets.MonthEnd(3)
Out[110]:
PeriodIndex(
    dtype='period[M]')

• Added experimental compatibility with openpyxl for versions >= 2.0. The DataFrame.to_excel method engine keyword now recognizes openpyxl1 and openpyxl2 which will explicitly require openpyxl v1 and v2 respectively, failing if the requested version is not available. The openpyxl engine is now a meta-engine that automatically uses whichever version of openpyxl is installed. (GH7177)

• DataFrame.fillna can now accept a DataFrame as a fill value (GH8377)

• Passing multiple levels to stack() will now work when multiple level numbers are passed (GH7660). See Reshaping by stacking and unstacking.

• set_names(), set_labels(), and set_levels() methods now take an optional level keyword argument to all modification of specific level(s) of a MultiIndex. Additionally set_names() now accepts a scalar string value when operating on an Index or on a specific level of a MultiIndex (GH7792)

In [111]: idx = pd.MultiIndex.from_product([[a], range(3), list("pqr")],
                             names=['foo', 'bar', 'baz'])
   .....:
   .....:

In [112]: idx.set_names('qux', level=0)
Out[112]:
MultiIndex(
    ['a', 0, 'p'],
    ['a', 0, 'q'],
    ['a', 0, 'r'],
    ['a', 1, 'p'],
    ['a', 1, 'q'],
    ['a', 1, 'r'],
    ['a', 2, 'p'],
    ['a', 2, 'q'],
    ['a', 2, 'r'],
    names=['qux', 'bar', 'baz'])

In [113]: idx.set_names(['qux', 'corge'], level=[0, 1])
Out[113]:
MultiIndex(
    ['a', 0, 'p'],
    ['a', 0, 'q'],
    ['a', 0, 'r'],
    ['a', 1, 'p'],
    ['a', 1, 'q'],
    (continues on next page)
• Index.isin now supports a level argument to specify which index level to use for membership tests (GH7892, GH7890)

In [1]: idx = pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c']])

In [2]: idx.values
Out[2]: array([(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')], dtype=object)

In [3]: idx.isin(['a', 'c', 'e'], level=1)
Out[3]: array([ True, False, True, True, False, True], dtype=bool)

• Index now supports duplicated and drop_duplicates. (GH4060)

In [116]: idx = pd.Index([1, 2, 3, 4, 1, 2])

In [117]: idx
Out[117]: Int64Index([1, 2, 3, 4, 1, 2], dtype='int64')

In [118]: idx.duplicated()
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

<table>
<thead>
<tr>
<th>Out[118]:</th>
<th>array([False, False, False, False, True, True])</th>
</tr>
</thead>
<tbody>
<tr>
<td>In [119]:</td>
<td>idx.drop_duplicates()</td>
</tr>
<tr>
<td>Out[119]:</td>
<td>Int64Index([1, 2, 3, 4], dtype='int64')</td>
</tr>
</tbody>
</table>

- add copy=True argument to pd.concat to enable pass through of complete blocks (GH8252)
- Added support for numpy 1.8+ data types (bool_, int_, float_, string_) for conversion to R dataframe (GH8400)

**Performance**

- Performance improvements in DatetimeIndex.__iter__ to allow faster iteration (GH7683)
- Performance improvements in Period creation (and PeriodIndex setitem) (GH5155)
- Improvements in Series.transform for significant performance gains (revised) (GH6496)
- Performance improvements in StataReader when reading large files (GH8040, GH8073)
- Performance improvements in StataWriter when writing large files (GH8079)
- Performance and memory usage improvements in multi-key groupby (GH8128)
- Performance improvements in groupby .agg and .apply where builtins max/min were not mapped to numpy/cythonized versions (GH7722)
- Performance improvement in writing to sql (to_sql) of up to 50% (GH8208).
- Performance benchmarking of groupby for large value of ngroups (GH6787)
- Performance improvement in CustomBusinessDay, CustomBusinessMonth (GH8236)
- Performance improvement for MultiIndex.values for multi-level indexes containing datetimes (GH8543)

**Bug fixes**

- Bug in pivot_table, when using margins and a dict aggfunc (GH8349)
- Bug in read_csv where squeeze=True would return a view (GH8217)
- Bug in checking of table name in read_sql in certain cases (GH7826).
- Bug in DataFrame.groupby where Grouper does not recognize level when frequency is specified (GH7885)
- Bug in multiindexes dtypes getting mixed up when DataFrame is saved to SQL table (GH8021)
- Bug in Series 0-division with a float and integer operand dtypes (GH7785)
- Bug in Series.astype(“unicode”) not calling unicode on the values correctly (GH7758)
- Bug in DataFrame.as_matrix() with mixed datetime64[ns] and timedelta64[ns] dtypes (GH7778)
- Bug in HDFStore.select_column() not preserving UTC timezone info when selecting a DatetimeIndex (GH7777)
- Bug in to_datetime when format='%Y%m%d' and coerce=True are specified, where previously an object array was returned (rather than a coerced time-series with NaT), (GH7930)
- Bug in DatetimeIndex and PeriodIndex in-place addition and subtraction cause different result from normal one (GH6527)
• Bug in adding and subtracting PeriodIndex with PeriodIndex raise TypeError (GH7741)
• Bug in combine_first with PeriodIndex data raises TypeError (GH3367)
• Bug in MultiIndex slicing with missing indexers (GH7866)
• Bug in MultiIndex slicing with various edge cases (GH8132)
• Regression in MultiIndex indexing with a non-scalar type object (GH7914)
• Bug in Timestamp comparisons with == and int64 dtype (GH8058)
• Bug in pickles contains DateOffset may raise AttributeError when normalize attribute is referred internally (GH7748)
• Bug in Panel when using major_xs and copy=False is passed (deprecation warning fails because of missing warnings) (GH8152).
• Bug in pickle deserialization that failed for pre-0.14.1 containers with dup items trying to avoid ambiguity when matching block and manager items, when there’s only one block there’s no ambiguity (GH7794)
• Bug in putting a PeriodIndex into a Series would convert to int64 dtype, rather than object of Periods (GH7932)
• Bug in HDFStore iteration when passing a where (GH8014)
• Bug in DataFrameGroupby.transform when transforming with a passed non-sorted key (GH8046, GH8430)
• Bug in repeated timeseries line and area plot may result in ValueError or incorrect kind (GH7733)
• Bug in inference in a MultiIndex with datetime.date inputs (GH7888)
• Bug in get where an IndexError would not cause the default value to be returned (GH7725)
• Bug in offsets.apply, rollforward and rollback may reset nanosecond (GH7697)
• Bug in offsets.apply, rollforward and rollback may raise AttributeError if Timestamp has dateutil tzinfo (GH7697)
• Bug in sorting a MultiIndex frame with a Float64Index (GH8017)
• Bug in inconsistent panel setitem with a rhs of a DataFrame for alignment (GH7763)
• Bug in is_superperiod and is_subperiod cannot handle higher frequencies than S (GH7760, GH7772, GH7803)
• Bug in 32-bit platforms with Series.shift (GH8129)
• Bug in PeriodIndex.unique returns int64 np.ndarray (GH7540)
• Bug in groupby.apply with a non-affecting mutation in the function (GH8467)
• Bug in DataFrame.reset_index which has MultiIndex contains PeriodIndex or DatetimeIndex with tz raises ValueError (GH7746, GH7793)
• Bug in DataFrame.plot with subplots=True may draw unnecessary minor xticks and yticks (GH7801)
• Bug in StataReader which did not read variable labels in 117 files due to difference between Stata documentation and implementation (GH7816)
• Bug in StataReader where strings were always converted to 244 characters-fixed width irrespective of underlying string size (GH7858)
• Bug in DataFrame.plot and Series.plot may ignore rot and fontsize keywords (GH7844)
• Bug in DatetimeIndex.value_counts doesn't preserve tz (GH7735)
• Bug in PeriodIndex.value_counts results in Int64Index (GH7735)
• Bug in DataFrame.join when doing left join on index and there are multiple matches (GH5391)

• Bug in GroupBy.transform() where int groups with a transform that didn’t preserve the index were incorrectly truncated (GH7972).

• Bug in groupby where callable objects without name attributes would take the wrong path, and produce a DataFrame instead of a Series (GH7929)

• Bug in groupby error message when a DataFrame grouping column is duplicated (GH7511)

• Bug in read_html where the infer_types argument forced coercion of date-likes incorrectly (GH7762, GH7032).

• Bug in Series.str.cat with an index which was filtered as to not include the first item (GH7857)

• Bug in Timestamp cannot parse nanosecond from string (GH7878)

• Bug in Timestamp with string offset and tz results incorrect (GH7833)

• Bug in tslib.tz_convert and tslib.tz_convert_single may return different results (GH7798)

• Bug in DatetimeIndex.intersection of non-overlapping timestamps with tz raises IndexError (GH7880)

• Bug in alignment with TimeOps and non-unique indexes (GH8363)

• Bug in GroupBy.filter() where fast path vs. slow path made the filter return a non scalar value that appeared valid but wasn’t (GH7870).

• Bug in date_range()/DatetimeIndex() when the timezone was inferred from input dates yet incorrect times were returned when crossing DST boundaries (GH7835, GH7901).

• Bug in to_excel() where a negative sign was being prepended to positive infinity and was absent for negative infinity (GH7949)

• Bug in area plot draws legend with incorrect alpha when stacked=True (GH8027)

• Period and PeriodIndex addition/subtraction with np.timedelta64 results in incorrect internal representations (GH7740)

• Bug in Holiday with no offset or observance (GH7987)

• Bug in DataFrame.to_latex formatting when columns or index is a MultiIndex (GH7982).

• Bug in DateOffset around Daylight Savings Time produces unexpected results (GH5175).

• Bug in DataFrame.shift where empty columns would throw ZeroDivisionError on numpy 1.7 (GH8019)

• Bug in installation where html_encoding/*.html wasn’t installed and therefore some tests were not running correctly (GH7927).

• Bug in read_html where bytes objects were not tested for in _read (GH7927).

• Bug in DataFrame.stack() when one of the column levels was a datelike (GH8039)

• Bug in broadcasting numpy scalars with DataFrame (GH8116)

• Bug in pivot_table performed with nameless index and columns raises KeyError (GH8103)

• Bug in DataFrame.plot(kind='scatter') draws points and errorbars with different colors when the color is specified by c keyword (GH8081)

• Bug in Float64Index where iat and at were not testing and were failing (GH8092).

• Bug in DataFrame.boxplot() where y-limits were not set correctly when producing multiple axes (GH7528, GH5517).

• Bug in read_csv where line comments were not handled correctly given a custom line terminator or delim_whitespace=True (GH8122).
- Bug in `read_html` where empty tables caused a `StopIteration` (GH7575)
- Bug in casting when setting a column in a same-dtype block (GH7704)
- Bug in accessing groups from a `GroupBy` when the original grouper was a tuple (GH8121).
- Bug in `.at` that would accept integer indexers on a non-integer index and do fallback (GH7814)
- Bug with `kde` plot and NaNs (GH8182)
- Bug in `GroupBy.count` with float32 data type were nan values were not excluded (GH8169).
- Bug with stacked barplots and NaNs (GH8175).
- Bug in resample with non evenly divisible offsets (e.g. ‘7s’) (GH8371)
- Bug in interpolation methods with the `limit` keyword when no values needed interpolating (GH7173).
- Bug where `col_space` was ignored in `DataFrame.to_string()` when `header=False` (GH8230).
- Bug with `DatetimeIndex.asof` incorrectly matching partial strings and returning the wrong date (GH8245).
- Bug in plotting methods modifying the global matplotlib rcParams (GH8242).
- Bug in `DataFrame.__setitem__` that caused errors when setting a dataframe column to a sparse array (GH8131)
- Bug where `DataFrame.boxplot()` failed when entire column was empty (GH8181).
- Bug with messed variables in `radviz` visualization (GH8199).
- Bug in interpolation methods with the `limit` keyword when no values needed interpolating (GH7173).
- Bug where `col_space` was ignored in `DataFrame.to_string()` when `header=False` (GH8230).
- Bug in `to_clipboard` that would clip long column data (GH8305)
- Bug in `DataFrame` terminal display: Setting `max_column/max_rows` to zero did not trigger auto-resizing of dfs to fit terminal width/height (GH7180).
- Bug in OLS where running with “cluster” and “nw_lags” parameters did not work correctly, but also did not throw an error (GH5884).
- Bug in `DataFrame.dropna` that interpreted non-existent columns in the subset argument as the ‘last column’ (GH8303)
- Bug in `Index.intersection` on non-monotonic non-unique indexes (GH8362).
- Bug in masked series assignment where mismatching types would break alignment (GH8387)
- Bug in `NDFrame.equals` gives false negatives with `dtype=object` (GH8437)
- Bug in assignment with indexer where type diversity would break alignment (GH8258)
- Bug in `NDFrame.loc` indexing when row/column names were lost when target was a list/ndarray (GH6552)
- Regression in `NDFrame.loc` indexing when rows/columns were converted to `Float64Index` if target was an empty list/ndarray (GH7774)
- Bug in `Series` that allows it to be indexed by a `DataFrame` which has unexpected results. Such indexing is no longer permitted (GH8444)
- Bug in item assignment of a `DataFrame` with MultiIndex columns where right-hand-side columns were not aligned (GH7655)
- Suppress FutureWarning generated by NumPy when comparing object arrays containing NaN for equality (GH7065)
- Bug in `DataFrame.eval()` where the dtype of the `not` operator (~) was not correctly inferred as `bool`
Contributors

A total of 80 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Schumacher +
- Adam Greenhall
- Andy Hayden
- Anthony O’Brien +
- Artemy Kolchinsky +
- Ben Schiller +
- Benedikt Sauer
- Benjamin Thureau +
- BorisVerk +
- Chris Reynolds +
- Chris Stoaefer +
- DSM
- Dav Clark +
- FragLegs +
- German Gomez-Herrero +
- Hsiaoming Yang +
- Huan Li +
- Hyungtae Kim +
- Isaac Slavitt +
- Jacob Schaer
- Jacob Wasserman +
- Jan Schulz
- Jeff Reback
- Jeff Tratner
- Jesse Farnham +
- Joe Bradish +
- Joerg Rittinger +
- John W. O’Brien
- Joris Van den Bossche
- Kevin Sheppard
- Kyle Meyer
- Max Chang +
- Michael Mueller
• Michael W Schatzow +
• Mike Kelly
• Mortada Mehyar
• Nathan Sanders +
• Nathan Typanski +
• Paul Masurel +
• Phillip Cloud
• Pietro Battiston
• RenzoBertocchi +
• Ross Petchler +
• Shahul Hameed +
• Shashank Agarwal +
• Stephan Hoyer
• Tom Augspurger
• TomAugspurger
• Tony Lorenzo +
• Wes Turner
• Wilfred Hughes +
• Yevgeniy Grechka +
• Yoshiki Vázquez Baeza +
• behzad nouri +
• benjamin
• bjonen +
• dlovell +
• dsm054
• hunterowens +
• immerrr
• ischwabacher
• jmorris0x0 +
• jnmclarty +
• jreback
• klonuo +
• lexical
• mcjcode +
• mtrbean +
• onesandzeroes
This is a minor release from 0.14.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

• Highlights include:
  – New methods `select_dtypes()` to select columns based on the dtype and `sem()` to calculate the standard error of the mean.
  – Support for dateutil timezones (see `docs`).
  – Support for ignoring full line comments in the `read_csv()` text parser.
  – New documentation section on `Options and Settings`.
  – Lots of bug fixes.

• `Enhancements`
• `API Changes`
• `Performance Improvements`
• `Experimental Changes`
• `Bug Fixes`
API changes

- Openpyxl now raises a ValueError on construction of the openpyxl writer instead of warning on pandas import (GH7284).
- For StringMethods.extract, when no match is found, the result - only containing NaN values - now also has dtype=object instead of float (GH7242).
- Period objects no longer raise a TypeError when compared using == with another object that isn't a Period. Instead when comparing a Period with another object using == if the other object isn't a Period False is returned. (GH7376)
- Previously, the behaviour on resetting the time or not in offsets.apply, rollforward and rollback operations differed between offsets. With the support of the normalize keyword for all offsets(see below) with a default value of False (preserve time), the behaviour changed for certain offsets (BusinessMonthBegin, MonthEnd, BusinessMonthEnd, CustomBusinessMonthEnd, BusinessYearBegin, LastWeekOfMonth, FY5253Quarter, LastWeekOfMonth, Easter):

```python
In [6]: from pandas.tseries import offsets

In [7]: d = pd.Timestamp('2014-01-01 09:00')

# old behaviour < 0.14.1
In [8]: d + offsets.MonthEnd()
Out[8]: pd.Timestamp('2014-01-31 00:00:00')

Starting from 0.14.1 all offsets preserve time by default. The old behaviour can be obtained with normalize=True

# new behaviour
In [1]: d + offsets.MonthEnd()
Out[1]: Timestamp('2014-01-31 09:00:00')

In [2]: d + offsets.MonthEnd(normalize=True)
Out[2]: Timestamp('2014-01-31 00:00:00')

Note that for the other offsets the default behaviour did not change.

- Add back #N/A N/A as a default NA value in text parsing. (regression from 0.12) (GH5521)
- Raise a TypeError on inplace-setting with a .where and a non np.nan value as this is inconsistent with a set-item expression like df[mask] = None (GH7656)

Enhancements

- Add dropna argument to value_counts and nunique (GH5569).
- Add select_dtypes() method to allow selection of columns based on dtype (GH7316). See the docs.
- All offsets supports the normalize keyword to specify whether offsets.apply, rollforward and rollback resets the time (hour, minute, etc) or not (default False, preserves time) (GH7156):

```python
import pandas.tseries.offsets as offsets
day = offsets.Day()
day.apply(pd.Timestamp("2014-01-01 09:00"))
```

(continues on next page)
day = offsets.Day(normalize=True)
day.apply(pd.Timestamp("2014-01-01 09:00"))

• PeriodIndex is represented as the same format as DatetimeIndex (GH7601)
• StringMethods now work on empty Series (GH7242)
• The file parsers read_csv and read_table now ignore line comments provided by the parameter comment, which accepts only a single character for the C reader. In particular, they allow for comments before file data begins (GH2685)
• Add NotImplemented for simultaneous use of chunksize and nrows for read_csv() (GH6774).
• Tests for basic reading of public S3 buckets now exist (GH7281).
• read_html now sports an encoding argument that is passed to the underlying parser library. You can use this to read non-ascii encoded web pages (GH7323).
• read_excel now supports reading from URLs in the same way that read_csv does. (GH6809)
• Support for dateutil timezones, which can now be used in the same way as pytz timezones across pandas. (GH4688)

In [3]: rng = pd.date_range(
                   ...: "3/6/2012 00:00", periods=10, freq="D", tz="dateutil/Europe/London"
                   ...: )
                   ...:
In [4]: rng.tz
Out[4]: tzfile('/usr/share/zoneinfo/Europe/London')

See the docs.
• Implemented sem (standard error of the mean) operation for Series, DataFrame, Panel, and Groupby (GH6897)
• Add nlargest and nsmallest to the Series groupby allowlist, which means you can now use these methods on a SeriesGroupBy object (GH7053).
• All offsets apply, rollforward and rollback can now handle np.datetime64, previously results in ApplyTypeError (GH7452)
• Period and PeriodIndex can contain NaT in its values (GH7485)
• Support pickling Series, DataFrame and Panel objects with non-unique labels along item axis (index, columns and items respectively) (GH7370).
• Improved inference of datetime/timedelta with mixed null objects. Regression from 0.13.1 in interpretation of an object Index with all null elements (GH7431)
Performance

- Improvements in dtype inference for numeric operations involving yielding performance gains for dtypes: int64, timedelta64, datetime64 (GH7223)
- Improvements in Series.transform for significant performance gains (GH6496)
- Improvements in DataFrame.transform with ufuncs and built-in grouper functions for significant performance gains (GH7383)
- Regression in groupby aggregation of datetime64 dtypes (GH7555)
- Improvements in MultiIndex.from_product for large iterables (GH7627)

Experimental

- pandas.io.data.Options has a new method, get_all_data method, and now consistently returns a Multi-Indexed DataFrame (GH5602)
- io.gbq.read_gbq and io.gbq.to_gbq were refactored to remove the dependency on the Google bq.py command line client. This submodule now uses httplib2 and the Google apiclient and oauth2client API client libraries which should be more stable and, therefore, reliable than bq.py. See the docs. (GH6937).

Bug fixes

- Bug in DataFrame.where with a symmetric shaped frame and a passed other of a DataFrame (GH7506)
- Bug in Panel indexing with a MultiIndex axis (GH7516)
- Regression in datetimelike slice indexing with a duplicated index and non-exact end-points (GH7523)
- Bug in setitem with list-of-lists and single vs mixed types (GH7551)
- Bug in time ops with non-aligned Series (GH7500)
- Bug in timedelta inference when assigning an incomplete Series (GH7592)
- Bug in groupby .nth with a Series and integer-like column name (GH7559)
- Bug in Series.get with a boolean accessor (GH7407)
- Bug in value_counts where NaT did not qualify as missing (NaN) (GH7423)
- Bug in to_timedelta that accepted invalid units and misinterpreted 'm/h' (GH7611, GH6423)
- Bug in line plot doesn’t set correct xlim if secondary_y=True (GH7459)
- Bug in grouped hist and scatter plots use old figsize default (GH7394)
- Bug in plotting subplots with DataFrame.plot, hist clears passed ax even if the number of subplots is one (GH7391).
- Bug in plotting subplots with DataFrame.boxplot with by kw raises ValueError if the number of subplots exceeds 1 (GH7391).
- Bug in subplots displays ticklabels and labels in different rule (GH5897)
- Bug in Panel.apply with a MultiIndex as an axis (GH7469)
- Bug in DatetimeIndex.insert doesn’t preserve name and tz (GH7299)
- Bug in DatetimeIndex.asobject doesn’t preserve name (GH7299)
- Bug in MultiIndex slicing with datetimelike ranges (strings and Timestamps), (GH7429)
• Bug in Index.min and max doesn’t handle nan and NaT properly (GH7261)
• Bug in PeriodIndex.min/max results in int (GH7609)
• Bug in resample where fill_method was ignored if you passed how (GH2073)
• Bug in TimeGrouper doesn’t exclude column specified by key (GH7227)
• Bug in DataFrame and Series bar and barh plot raises TypeError when bottom and left keyword is specified (GH7226)
• Bug in DataFrame.hist raises TypeError when it contains non numeric column (GH7277)
• Bug in Index.delete does not preserve name and freq attributes (GH7302)
• Bug in DataFrame.query() where local string variables with the @ sign were being treated as temporaries attempting to be deleted (GH7300).
• Bug in Float64Index which didn’t allow duplicates (GH7149).
• Bug in DataFrame.replace() where truthy values were being replaced (GH7140).
• Bug in StringMethods.extract() where a single match group Series would use the matcher’s name instead of the group name (GH7313).
• Bug in isnull() when mode.use_inf_as_null == True where isnull wouldn’t test True when it encountered an inf/-inf (GH7315).
• Bug in inferred_freq results in None for eastern hemisphere timezones (GH7310)
• Bug in Easter returns incorrect date when offset is negative (GH7195)
• Bug in broadcasting with .div. integer dtypes and divide-by-zero (GH7325)
• Bug in CustomBusinessDay.apply raises NameError when np.datetime64 object is passed (GH7196)
• Bug in MultiIndex.append. concat and pivot_table don’t preserve timezone (GH6606)
• Bug in .loc with a list of indexers on a single-multi index level (that is not nested) (GH7349)
• Bug in Series.map when mapping a dict with tuple keys of different lengths (GH7333)
• Bug all StringMethods now work on empty Series (GH7242)
• Fix delegation of read_sql to read_sql_query when query does not contain ‘select’ (GH7324).
• Bug where a string column name assignment to a DataFrame with a Float64Index raised a TypeError during a call to np.isnan (GH7366).
• Bug where NDFrame.replace() didn’t correctly replace objects with Period values (GH7379).
• Bug in .ix getitem should always return a Series (GH7150)
• Bug in MultiIndex slicing with incomplete indexers (GH7399)
• Bug in MultiIndex slicing with a step in a sliced level (GH7400)
• Bug where negative indexers in DatetimeIndex were not correctly sliced (GH7408)
• Bug where NaT wasn’t repr’d correctly in a MultiIndex (GH7406, GH7409).
• Bug where bool objects were converted to nan in convert_objects (GH7416).
• Bug in quantile ignoring the axis keyword argument (GH7306)
• Bug where nanops._maybe_null_out doesn’t work with complex numbers (GH7353)
• Bug in several nanops functions when axis==0 for 1-dimensional nan arrays (GH7354)
• Bug where nanops.nanmedian doesn’t work when axis=None (GH7352)
• Bug where nanops._has_infs doesn't work with many dtypes (GH7357)
• Bug in StataReader.data where reading a 0-observation dta failed (GH7369)
• Bug in StataReader when reading Stata 13 (117) files containing fixed width strings (GH7360)
• Bug in StataWriter where encoding was ignored (GH7286)
• Bug in DatetimeIndex comparison doesn't handle NaT properly (GH7529)
• Bug in passing input with tzinfo to some offsets apply, rollforward or rollback resets tzinfo or raises ValueError (GH7465)
• Bug in DatetimeIndex.to_period, PeriodIndex.asobject, PeriodIndex.to_timestamp doesn't preserve name (GH7485)
• Bug in DatetimeIndex.to_period and PeriodIndex.to_timestamp handle NaT incorrectly (GH7228)
• Bug in offsets.apply, rollforward and rollback may return normal datetime (GH7502)
• Bug in resample raises ValueError when target contains NaT (GH7227)
• Bug in Timestamp.tz_localize resets nanosecond info (GH7534)
• Bug in DatetimeIndex.asobject raises ValueError when it contains NaT (GH7539)
• Bug in Timestamp._new__ doesn't preserve nanosecond properly (GH7610)
• Bug in Index.astype(float) where it would return an object dtype Index (GH7464).
• Bug in DataFrame.reset_index loses tz (GH3950)
• Bug in DatetimeIndex.freqstr raises AttributeError when freq is None (GH7606)
• Bug in GroupBy.size created by TimeGrouper raises AttributeError (GH7453)
• Bug in single column bar plot is misaligned (GH7498).
• Bug in area plot with tz-aware time series raises ValueError (GH7471)
• Bug in non-monotonic Index.union may preserve name incorrectly (GH7458)
• Bug in DatetimeIndex.intersection doesn't preserve timezone (GH4690)
• Bug in rolling_var where a window larger than the array would raise an error (GH7297)
• Bug with last plotted timeseries dictating xlim (GH2960)
• Bug with secondary_y axis not being considered for timeseries xlim (GH3490)
• Bug in Float64Index assignment with a non scalar indexer (GH7586)
• Bug in pandas.core.strings.str_contains does not properly match in a case insensitive fashion when regex=False and case=False (GH7505)
• Bug in expanding_cov, expanding_corr, rolling_cov, and rolling_corr for two arguments with mismatched index (GH7512)
• Bug in to_sql taking the boolean column as text column (GH7678)
• Bug in grouped hist doesn't handle rot kw and sharex kw properly (GH7234)
• Bug in .loc performing fallback integer indexing with object dtype indices (GH7496)
• Bug (regression) in PeriodIndex constructor when passed Series objects (GH7701).
Contributors

A total of 46 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Andrew Rosenfeld
- Andy Hayden
- Benjamin Adams +
- Benjamin M. Gross +
- Brian Quistorff +
- Brian Wignall +
- DSM
- Daniel Waebber
- David Bew +
- David Stephens
- Jacob Schaefer
- Jan Schulz
- John David Reaver
- John W. O’Brien
- Joris Van den Bossche
- Julien Danjou +
- K.-Michael Aye
- Kevin Sheppard
- Kyle Meyer
- Matt Wittmann
- Matthew Brett +
- Michael Mueller +
- Mortada Mehyar
- Phillip Cloud
- Rob Levy +
- Schaefer, Jacob C +
- Stephan Hoyer
- Thomas Kluyver
- Todd Jennings
- Tom Augspurger
- TomAugspurger
- bwignall
- clham
This is a major release from 0.13.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- **Highlights include:**
  - Officially support Python 3.4
  - SQL interfaces updated to use `sqlalchemy`, See [Here](#).
  - Display interface changes, See [Here](#).
  - MultiIndexing Using Slicers, See [Here](#).
  - Ability to join a singly-indexed DataFrame with a MultiIndexed DataFrame, see [Here](#).
  - More consistency in groupby results and more flexible groupby specifications, See [Here](#).
  - Holiday calendars are now supported in `CustomBusinessDay`, see [Here](#).
  - Several improvements in plotting functions, including: hexbin, area and pie plots, see [Here](#).
  - Performance doc section on I/O operations, See [Here](#).

- **Other Enhancements**
- **API Changes**
- **Text Parsing API Changes**
- **Groupby API Changes**
- **Performance Improvements**
- **Prior Deprecations**
- **Deprecations**
- **Known Issues**
- **Bug Fixes**
**Warning:** In 0.14.0 all `NDFrame` based containers have undergone significant internal refactoring. Before that each block of homogeneous data had its own labels and extra care was necessary to keep those in sync with the parent container’s labels. This should not have any visible user/API behavior changes (GH6745)

### API changes

- `read_excel` uses 0 as the default sheet (GH6573)
- `iloc` will now accept out-of-bounds indexers for slices, e.g. a value that exceeds the length of the object being indexed. These will be excluded. This will make pandas conform more with python/numpy indexing of out-of-bounds values. A single indexer that is out-of-bounds and drops the dimensions of the object will still raise `IndexError` (GH6296, GH6299). This could result in an empty axis (e.g. an empty DataFrame being returned)

```python
In [1]: dfl = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))
In [2]: dfl
Out[2]:
   A   B
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569

[5 rows x 2 columns]
In [3]: dfl.iloc[:, 2:3]
Out[3]:
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

[5 rows x 0 columns]
In [4]: dfl.iloc[:, 1:3]
Out[4]:
   B
0 -0.282863
1 -1.135632
2 -0.173215
3 -1.044236
4 -2.104569

[5 rows x 1 columns]
In [5]: dfl.iloc[4:6]
Out[5]:
   A   B
4 -0.861849 -2.104569

[1 rows x 2 columns]
```

These are out-of-bounds selections
>>> dfl.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds

>>> dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds

• Slicing with negative start, stop & step values handles corner cases better (GH6531):
  - df.iloc[:len(df)] is now empty
  - df.iloc[len(df):len(df)-1] now enumerates all elements in reverse

• The DataFrame.interpolate() keyword downcast default has been changed from infer to None. This is to preserve the original dtype unless explicitly requested otherwise (GH6290).

• When converting a dataframe to HTML it used to return Empty DataFrame. This special case has been removed, instead a header with the column names is returned (GH6062).

• Series and Index now internally share more common operations, e.g. factorize(), nunique(), value_counts() are now supported on Index types as well. The Series.weekday property from is removed from Series for API consistency. Using a DatetimeIndex/PeriodIndex method on a Series will now raise a TypeError. (GH4551, GH4056, GH5519, GH6380, GH7206).

• Add is_month_start, is_month_end, is_quarter_start, is_quarter_end, is_year_start, is_year_end accessors for DateTimeIndex / Timestamp which return a boolean array of whether the timestamp(s) are at the start/end of the month/quarter/year defined by the frequency of the DateTimeIndex / Timestamp (GH4565, GH6998)

• Local variable usage has changed in pandas.eval()/DataFrame.eval()/DataFrame.query() (GH5987). For the DataFrame methods, two things have changed
  - Column names are now given precedence over locals
  - Local variables must be referred to explicitly. This means that even if you have a local variable that is not a column you must still refer to it with the '@' prefix.
  - You can have an expression like df.query('@a < a') with no complaints from pandas about ambiguity of the name a.
  - The top-level pandas.eval() function does not allow you use the '@' prefix and provides you with an error message telling you so.
  - NameResolutionError was removed because it isn’t necessary anymore.

• Define and document the order of column vs index names in query/eval (GH6676)

• concat will now concatenate mixed Series and DataFrames using the Series name or numbering columns as needed (GH2385). See the docs

• Slicing and advanced/boolean indexing operations on Index classes as well as Index.delete() and Index.drop() methods will no longer change the type of the resulting index (GH6440, GH7040)
Previously, the above operation would return `Int64Index`. If you’d like to do this manually, use `Index.astype()`

```
In [9]: i[[0, 1, 2]].astype(np.int_)
Out[9]: Int64Index([1, 2, 3], dtype='int64')
```

• `set_index` no longer converts MultiIndexes to an Index of tuples. For example, the old behavior returned an Index in this case (GH6459):

```
# Old behavior, casted MultiIndex to an Index
In [10]: tuple_ind
Out[10]: Index([('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')], dtype='object')

In [11]: df_multi.set_index(tuple_ind)
Out[11]:
    0  1
(a, c) 0.471435 -1.190976
(a, d) 1.432707 -0.312652
(b, c) -0.720589  0.887163
(b, d)  0.859588 -0.636524

[4 rows x 2 columns]

# New behavior
In [12]: mi
Out[12]: MultiIndex([('a', 'c'),
                     ('a', 'd'),
                     ('b', 'c'),
                     ('b', 'd')],
                    )

In [13]: df_multi.set_index(mi)
Out[13]:
    0  1
 a c  0.471435 -1.190976
 d  1.432707 -0.312652
 b c -0.720589  0.887163
 d  0.859588 -0.636524

[4 rows x 2 columns]
```

This also applies when passing multiple indices to `set_index`:

```
# Old output, 2-level MultiIndex of tuples
In [14]: df_multi.set_index([df_multi.index, df_multi.index])
Out[14]:
    0  1
(a, c) (a, c) 0.471435 -1.190976
(a, d) (a, d) 1.432707 -0.312652
(b, c) (b, c) -0.720589  0.887163
(b, d) (b, d)  0.859588 -0.636524

[4 rows x 2 columns]
```

(continues on next page)
# New output, 4-level MultiIndex

```
In [15]: df_multi.set_index([df_multi.index, df_multi.index])
Out[15]:
    0  1
a c c  0.471435 -1.190976
a d d  1.432707 -0.312652
b c c -0.720589  0.887163
b d d  0.859588 -0.636524
[4 rows x 2 columns]
```

- `pairwise` keyword was added to the statistical moment functions `rolling_cov`, `rolling_corr`, `ewmcov`, `ewmcorr`, `expanding_cov`, `expanding_corr` to allow the calculation of moving window covariance and correlation matrices (GH4950). See [Computing rolling pairwise covariances and correlations](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Timeseries.rolling.html) in the docs.

```
In [1]: df = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'))
In [4]: covs = pd.rolling_cov(df[['A', 'B', 'C']],
                          df[['B', 'C', 'D']],
                          5,
                          pairwise=True)
In [5]: covs[df.index[-1]]
Out[5]:
   B    C    D
  A  0.035310  0.326593 -0.505430
  B  0.137748 -0.006888 -0.005383
  C -0.006888  0.861040  0.028762
```

- `Series.iteritems()` is now lazy (returns an iterator rather than a list). This was the documented behavior prior to 0.14. (GH6760)
- Added `nunique` and `value_counts` functions to `Index` for counting unique elements. (GH6734)
- `stack` and `unstack` now raise a `ValueError` when the `level` keyword refers to a non-unique item in the `Index` (previously raised a `KeyError`). (GH6738)
- `drop` unused order argument from `Series.sort`; args now are in the same order as `Series.order`; add `na_position` arg to conform to `Series.order` (GH6847)
- default sorting algorithm for `Series.order` is now quicksort, to conform with `Series.sort` (and numpy defaults)
- add `inplace` keyword to `Series.order/sort` to make them inverses (GH6859)
- `DataFrame.sort` now places NaNs at the beginning or end of the sort according to the `na_position` parameter. (GH3917)
- accept `TextFileReader` in `concat`, which was affecting a common user idiom (GH6583), this was a regression from 0.13.1
- Added `factorize` functions to `Index` and `Series` to get indexer and unique values (GH7090)
- `describe` on a `DataFrame` with a mix of Timestamp and string like objects returns a different `Index` (GH7088). Previously the index was unintentionally sorted.
• Arithmetic operations with only bool dtypes now give a warning indicating that they are evaluated in Python space for +, -, and * operations and raise for all others (GH7011, GH6762, GH7015, GH7210)

```python
>>> x = pd.Series(np.random.rand(10) > 0.5)
>>> y = True
>>> x + y  # warning generated: should do x | y instead
UserWarning: evaluating in Python space because the '+’ operator is not supported by numexpr for the bool dtype, use '|' instead
>>> x / y  # this raises because it doesn’t make sense
NotImplementedError: operator '/' not implemented for bool dtypes
```

• In HDFStore, select_as_multiple will always raise a KeyError, when a key or the selector is not found (GH6177)
• df['col'] = value and df.loc[:, 'col'] = value are now completely equivalent; previously the .loc would not necessarily coerce the dtype of the resultant series (GH6149)
• dtypes and ftypes now return a series with dtype=object on empty containers (GH5740)
• df.to_csv will now return a string of the CSV data if neither a target path nor a buffer is provided (GH6061)
• pd.infer_freq() will now raise a TypeError if given an invalid Series/Index type (GH6407, GH6463)
• A tuple passed to DataFrame.sort_index will be interpreted as the levels of the index, rather than requiring a list of tuple (GH4370)
• all offset operations now return Timestamp types (rather than datetime), Business/Week frequencies were incorrect (GH4069)
• to_excel now converts np.inf into a string representation, customizable by the inf_rep keyword argument (Excel has no native inf representation) (GH6782)
• Replace pandas.compat.scipy.scoreatpercentile with numpy.percentile (GH6810)
• .quantile on a datetime[ns] series now returns Timestamp instead of np.datetime64 objects (GH6810)
• change AssertionError to TypeError for invalid types passed to concat (GH6583)
• Raise a TypeError when DataFrame is passed an iterator as the data argument (GH5357)

**Display changes**

• The default way of printing large DataFrames has changed. DataFrames exceeding max_rows and/or max_columns are now displayed in a centrally truncated view, consistent with the printing of a pandas.Series (GH5603).

In previous versions, a DataFrame was truncated once the dimension constraints were reached and an ellipse (...) signaled that part of the data was cut off.
In [1]: import pandas as pd
In [2]: import numpy as np
In [3]: pd.options.display.max_rows = 6
In [4]: pd.options.display.max_columns = 6
In [5]: index = pd.DatetimeIndex(start='2001-01-01', freq='D', periods=10)
In [6]: pd.DataFrame(np.arange(10*10).reshape((10,10)), index=index)
Out[6]:
    0  1  2  3  4  5
2001-01-01  0  1  2  3  4  5 ...
2001-01-02 10 11 12 13 14 15 ...
2001-01-03 20 21 22 23 24 25 ...
2001-01-04 30 31 32 33 34 35 ...
2001-01-05 40 41 42 43 44 45 ...
2001-01-06 50 51 52 53 54 55 ...
...         ...    ...    ......

[10 rows x 10 columns]

In the current version, large DataFrames are centrally truncated, showing a preview of head and tail in both dimensions.

In [24]: pd.DataFrame(np.arange(10*10).reshape((10,10)), index=index)
Out[24]:
      0   1   2   ...   7   8   9
2001-01-01  0   1   2   ...   7   8   9
2001-01-02 10  11  12   ...  17  18  19
2001-01-03 20  21  22   ...  27  28  29
...        ...    ...    ...    ...    ...
2001-01-08 70  71  72   ...  77  78  79
2001-01-09 80  81  82   ...  87  88  89
2001-01-10 90  91  92   ...  97  98  99

[10 rows x 10 columns]

• allow option 'truncate' for display.show_dimensions to only show the dimensions if the frame is truncated (GH6547).

The default for display.show_dimensions will now be truncate. This is consistent with how Series display length.

In [16]: dfd = pd.DataFrame(np.arange(25).reshape(-1, 5),
                          index=[0, 1, 2, 3, 4],
                          columns=[0, 1, 2, 3, 4])
# show dimensions since this is truncated
In [17]: with pd.option_context('display.max_rows', 2, 'display.max_columns', 2,
....: 'display.show_dimensions', 'truncate'):
....:  print(dfd)
....:
   0  ...  4
0   0  ...  4
..   ..  ...  ..
4   20  ...  24

[5 rows x 5 columns]

# will not show dimensions since it is not truncated
In [18]: with pd.option_context('display.max_rows', 10, 'display.max_columns', 40,
....:                         'display.show_dimensions', 'truncate'):
....:  print(dfd)
....:
   0   1   2   3   4
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24

• Regression in the display of a MultiIndexed Series with display.max_rows is less than the length of the series (GH7101)

• Fixed a bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the large_repr set to ‘info’ (GH7105)

• The verbose keyword in DataFrame.info(), which controls whether to shorten the info representation, is now None by default. This will follow the global setting in display.max_info_columns. The global setting can be overridden with verbose=True or verbose=False.

• Fixed a bug with the info repr not honoring the display.max_info_columns setting (GH6939)

• Offset/freq info now in Timestamp __repr__ (GH4553)

Text parsing API changes

read_csv()/read_table() will now be noisier w.r.t invalid options rather than falling back to the PythonParser.

• Raise ValueError when sep specified with delim_whitespace=True in read_csv()/read_table() (GH6607)

• Raise ValueError when engine='c' specified with unsupported options in read_csv()/read_table() (GH6607)

• Raise ValueError when fallback to python parser causes options to be ignored (GH6607)

• Produce ParserWarning on fallback to python parser when no options are ignored (GH6607)

• Translate sep='\s+' to delim_whitespace=True in read_csv()/read_table() if no other C-unsupported options specified (GH6607)
GroupBy API changes

More consistent behavior for some groupby methods:

- groupby head and tail now act more like filter rather than an aggregation:

```python
In [19]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
In [20]: g = df.groupby('A')
In [21]: g.head(1)  # filters DataFrame
Out[21]:
   A  B
0  1  2
2  5  6
[2 rows x 2 columns]
In [22]: g.apply(lambda x: x.head(1))  # used to simply fall-through
Out[22]:
   A  B
A  1  0  1  2
   5  2  5  6
[2 rows x 2 columns]
```

- groupby head and tail respect column selection:

```python
In [23]: g[['B']].head(1)
Out[23]:
   B
0  2
2  6
[2 rows x 1 columns]
```

- groupby nth now reduces by default; filtering can be achieved by passing as_index=False. With an optional dropna argument to ignore NaN. See the docs.

Reducing

```python
In [24]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
In [25]: g = df.groupby('A')
In [26]: g.nth(0)
Out[26]:
   B
A
1  NaN
5  6.0
[2 rows x 1 columns]
```
# this is equivalent to g.first()
In [27]: g.nth(0, dropna='any')
Out[27]:
   A     B
0  1  4.0
5  6.0
[2 rows x 1 columns]

# this is equivalent to g.last()
In [28]: g.nth(-1, dropna='any')
Out[28]:
   A     B
A  1  4.0
5  6.0
[2 rows x 1 columns]

Filtering

In [29]: gf = df.groupby('A', as_index=False)
In [30]: gf.nth(0)
Out[30]:
   A   B
0  1  NaN
2  5  6.0
[2 rows x 2 columns]
In [31]: gf.nth(0, dropna='any')
Out[31]:
   A   B
A  1  4.0
5  6.0
[2 rows x 2 columns]

• groupby will now not return the grouped column for non-cython functions (GH5610, GH5614, GH6732), as it's already the index

In [32]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])
In [33]: g = df.groupby('A')
In [34]: g.count()
Out[34]:
   B
   1
0  0
1  1
5  2

A
1 1
5 2
[2 rows x 1 columns]

In [35]: g.describe()
Out[35]:

    B
    count mean std  min  25%  50%  75%  max
A
1  1.0  4.0 NaN  4.0  4.0  4.0  4.0
5  2.0  7.0 1.414214  6.0  6.5  7.0  7.5

[2 rows x 8 columns]

• passing as_index will leave the grouped column in-place (this is not change in 0.14.0)

In [36]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])

In [37]: g = df.groupby('A', as_index=False)

In [38]: g.count()
Out[38]:

    A  B
0  1  1
1  5  2

[2 rows x 2 columns]

In [39]: g.describe()
Out[39]:

    A  B
    count mean std  min  25%  50%  75%  max count mean std  min  25%  50%  75%  max
0   2.0  1.0  0.0  1.0  1.0  1.0  1.0  4.0   NaN  4.0  4.0  4.0  4.0  4.0
1   2.0  5.0  0.0  5.0  5.0  5.0  5.0  2.0  7.0  1.414214  6.0  6.5  7.0  7.5

[2 rows x 16 columns]

• Allow specification of a more complex groupby via pd.Grouper, such as grouping by a Time and a string field simultaneously. See the docs. (GH3794)

• Better propagation/preservation of Series names when performing groupby operations:
  – SeriesGroupBy.agg will ensure that the name attribute of the original series is propagated to the result (GH6265).
  – If the function provided to GroupBy.apply returns a named series, the name of the series will be kept as the name of the column index of the DataFrame returned by GroupBy.apply (GH6124). This facilitates
DataFrame.stack operations where the name of the column index is used as the name of the inserted column containing the pivoted data.

SQL

The SQL reading and writing functions now support more database flavors through SQLAlchemy (GH2717, GH4163, GH5950, GH6292). All databases supported by SQLAlchemy can be used, such as PostgreSQL, MySQL, Oracle, Microsoft SQL server (see documentation of SQLAlchemy on included dialects).

The functionality of providing DBAPI connection objects will only be supported for sqlite3 in the future. The 'mysql' flavor is deprecated.

The new functions read_sql_query() and read_sql_table() are introduced. The function read_sql() is kept as a convenience wrapper around the other two and will delegate to specific function depending on the provided input (database table name or sql query).

In practice, you have to provide a SQLAlchemy engine to the sql functions. To connect with SQLAlchemy you use the create_engine() function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For an in-memory sqlite database:

```python
In [40]: from sqlalchemy import create_engine

# Create your connection.
In [41]: engine = create_engine('sqlite:///memory:')</n```

This engine can then be used to write or read data to/from this database:

```python
In [42]: df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})
In [43]: df.to_sql('db_table', engine, index=False)
Out[43]: 3
```

You can read data from a database by specifying the table name:

```python
In [44]: pd.read_sql_table('db_table', engine)
Out[44]:
   A  B
0  1  a
1  2  b
2  3  c
[3 rows x 2 columns]
```

or by specifying a sql query:

```python
In [45]: pd.read_sql_query('SELECT * FROM db_table', engine)
Out[45]:
   A  B
0  1  a
1  2  b
2  3  c
[3 rows x 2 columns]
```

Some other enhancements to the sql functions include:
• support for writing the index. This can be controlled with the `index` keyword (default is True).
• specify the column label to use when writing the index with `index_label`.
• specify string columns to parse as datetimes with the `parse_dates` keyword in `read_sql_query()` and `read_sql_table()`.

**Warning:** Some of the existing functions or function aliases have been deprecated and will be removed in future versions. This includes: `tquery, uquery, read_frame, frame_query, write_frame`.

**Warning:** The support for the ‘mysql’ flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines (GH6900).

### Multi-indexing using slicers

In 0.14.0 we added a new way to slice MultiIndexed objects. You can slice a MultiIndex by providing multiple indexers. You can provide any of the selectors as if you are indexing by label, see Selection by Label, including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of that level. You do not need to specify all the deeper levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

See the docs See also issues (GH6134, GH4036, GH3057, GH2598, GH5641, GH7106)

**Warning:** You should specify all axes in the `.loc` specifier, meaning the indexer for the `index` and for the `columns`. Their are some ambiguous cases where the passed indexer could be mis-interpreted as indexing both axes, rather than into say the MultiIndex for the rows.

You should do this:

```python
>>> df.loc[(slice('A1', 'A3'), ...), :]  # noqa: E901
```

rather than this:

```python
>>> df.loc[(slice('A1', 'A3'), ...)]  # noqa: E901
```

**Warning:** You will need to make sure that the selection axes are fully lexsorted!

```python
In [46]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:
```

```python
In [47]: index = pd.MultiIndex.from_product([mklbl('A', 4),
   ....:     mklbl('B', 2),
   ....:     mklbl('C', 4),
```

(continues on next page)
In [48]: columns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'), ('b', 'foo'), ('b', 'bah')], names=['lvl0', 'lvl1'])

In [49]: df = pd.DataFrame(np.arange(len(index) * len(columns)).reshape((len(index), len(columns))), index=index, columns=columns).sort_index().sort_index(axis=1)

In [50]: df
Out[50]:
lvl0    a    b
lvl1    bar  foo  bah  foo
A0  B0  C0  D0   1    0    3    2
     D1   5    4    7    6
     C1  D0   9    8   11   10
     D1  13   12   15   14
     C2  D0  17   16   19   18
     D1  237  236  239  238
     B1  C1  D0  241  240  243  242
     D1  245  244  247  246
     C3  D0  249  248  251  250
     D1  253  252  255  254
     C2  D0  249  248  251  250
     D1  253  252  255  254
     B1  C1  D0  237  236  239  238
     D1  241  240  243  242
     C3  D0  249  248  251  250
     D1  253  252  255  254

[64 rows x 4 columns]

Basic MultiIndex slicing using slices, lists, and labels.

In [51]: df.loc[(slice('A1', 'A3'), slice(None), ['C1', 'C3']), :]
Out[51]:
lvl0    a    b
lvl1  bar  foo  bah  foo
A1  B0  C1  D0  73    72    75    74
     D1  77    76    79    78
     C3  D0  89    88    91    90
     D1  93    92    95    94
     B1  C1  D0 105   104   107   106
     D1  221  220  223  222
     B1  C1  D0 233  232  235  234
     D1  237  236  239  238
     C3  D0  249  248  251  250
     D1  253  252  255  254

[24 rows x 4 columns]

You can use a pd.IndexSlice to shortcut the creation of these slices.
In [52]: idx = pd.IndexSlice

In [53]: df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[53]:

lvl0       a   b
lvl1    foo   foo
A0  B0  C1  D0  8  10
   D1  12  14
   C3  D0  24  26
   D1  28  30
B1  C1  D0  40  42
... ... ...
A3  B0  C3  D1  220 222
B1  C1  D0  232 234
   D1  236 238
   C3  D0  248 250
   D1  252 254

[32 rows x 2 columns]

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

In [54]: df.loc['A1', (slice(None), 'foo')]
Out[54]:

lvl0       a   b
lvl1    foo   foo
B0  C0  D0  64  66
   D1  68  70
   C1  D0  72  74
   D1  76  78
   C2  D0  80  82
... ... ...
B1  C1  D1  108 110
   C2  D0  112 114
   D1  116 118
   C3  D0  120 122
   D1  124 126

[16 rows x 2 columns]

In [55]: df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[55]:

lvl0       a   b
lvl1    foo   foo
A0  B0  C1  D0  8  10
   D1  12  14
   C3  D0  24  26
   D1  28  30
B1  C1  D0  40  42
... ... ...
A3  B0  C3  D1  220 222
B1  C1  D0  232 234
   D1  236 238
Using a boolean indexer you can provide selection related to the values.

```python
In [56]: mask = df[['a', 'foo']] > 200

In [57]: df.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[57]:
   lvl0  a  b  
  lvl1 foo foo
   A3  B0  C1  D1  204  206
     C3  D0  216  218
     D1  220  222
   B1  C1  D0  232  234
     D1  236  238
     C3  D0  248  250
     D1  252  254
[7 rows x 2 columns]
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```python
In [58]: df.loc(axis=0)[[:, :, ['C1', 'C3']]
Out[58]:
   lvl0  a  b
  lvl1 bar foo bah foo
   A0  B0  C1  D0  9  8  11  10
     C3  D0  25  24  27  26
     D1  29  28  31  30
   B1  C1  D0  41  40  43  42
     D1  45  44  47  46
... ... ... ... ...
   A3  B0  C3  D1  221  220  223  222
   B1  C1  D0  233  232  235  234
     D1  237  236  239  238
     C3  D0  249  248  251  250
     D1  253  252  255  254
[32 rows x 4 columns]
```

Furthermore you can set the values using these methods

```python
In [59]: df2 = df.copy()

In [60]: df2.loc(axis=0)[[:, :, ['C1', 'C3']]] = -10

In [61]: df2
Out[61]:
   lvl0  a  b
  lvl1 bar foo bah foo
   A0  B0  C1  D0  9  8  11  10
     C3  D0  25  24  27  26
     D1  29  28  31  30
   B1  C1  D0  41  40  43  42
     D1  45  44  47  46
... ... ... ... ...
   A3  B0  C3  D1  221  220  223  222
   B1  C1  D0  233  232  235  234
     D1  237  236  239  238
     C3  D0  249  248  251  250
     D1  253  252  255  254
```

(continues on next page)
You can use a right-hand-side of an alignable object as well.

```python
In [62]: df2 = df.copy()
In [63]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000
In [64]: df2
Out[64]:
lvl0    a    b
lvl1 bar  foo  bah  foo
A0 B0 C0 D0  1  0  3  2
  D1   5  4  7  6
  C1 D0 -10 -10 -10 -10
  D1 -10 -10 -10 -10
  C2 D0  17  16  19  18
  ... ... ... ... ...
A3 B1 C1 D1 -10 -10 -10 -10
  C2 D0 241 240 243 242
  D1 245 244 247 246
  C3 D0 -10 -10 -10 -10
  D1 -10 -10 -10 -10
[64 rows x 4 columns]
```

Plotting

- Hexagonal bin plots from `DataFrame.plot` with `kind='hexbin'` (GH5478), See the docs.
- `DataFrame.plot` and `Series.plot` now supports area plot with specifying `kind='area'` (GH6656), See the docs
- Pie plots from `Series.plot` and `DataFrame.plot` with `kind='pie'` (GH6976), See the docs.
- Plotting with Error Bars is now supported in the `.plot` method of `DataFrame` and `Series` objects (GH3796, GH6834), See the docs.
- `DataFrame.plot` and `Series.plot` now support a `table` keyword for plotting `matplotlib.Table`, See the docs. The table keyword can receive the following values.
  - `False`: Do nothing (default).
pandas: powerful Python data analysis toolkit, Release 1.4.2

- True: Draw a table using the DataFrame or Series called plot method. Data will be transposed to meet matplotlib’s default layout.
- DataFrame or Series: Draw matplotlib.table using the passed data. The data will be drawn as displayed in print method (not transposed automatically). Also, helper function pandas.tools.plotting.table is added to create a table from DataFrame and Series, and add it to an matplotlib.Axes.

• plot(legend='reverse') will now reverse the order of legend labels for most plot kinds. (GH6014)
• Line plot and area plot can be stacked by stacked=True (GH6656)
• Following keywords are now acceptable for DataFrame.plot() with kind='bar' and kind='barh':
  - width: Specify the bar width. In previous versions, static value 0.5 was passed to matplotlib and it cannot be overwritten. (GH6604)
  - align: Specify the bar alignment. Default is center (different from matplotlib). In previous versions, pandas passes align='edge' to matplotlib and adjust the location to center by itself, and it results align keyword is not applied as expected. (GH4525)
  - position: Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1(right/top-end). Default is 0.5 (center). (GH6604)

Because of the default align value changes, coordinates of bar plots are now located on integer values (0.0, 1.0, 2.0 ...). This is intended to make bar plot be located on the same coordinates as line plot. However, bar plot may differs unexpectedly when you manually adjust the bar location or drawing area, such as using set_xlim, set_ylim, etc. In this cases, please modify your script to meet with new coordinates.

• The parallel_coordinates() function now takes argument color instead of colors. A FutureWarning is raised to alert that the old colors argument will not be supported in a future release. (GH6956)
• The parallel_coordinates() and andrews_curves() functions now take positional argument frame instead of data. A FutureWarning is raised if the old data argument is used by name. (GH6956)
• DataFrame.boxplot() now supports layout keyword (GH6769)
• DataFrame.boxplot() has a new keyword argument, return_type. It accepts 'dict', 'axes', or 'both', in which case a namedtuple with the matplotlib axes and a dict of matplotlib Lines is returned.

Prior version deprecations/changes

There are prior version deprecations that are taking effect as of 0.14.0.

• Remove DateRange in favor of DatetimeIndex (GH6816)
• Remove column keyword from DataFrame.sort (GH4370)
• Remove precision keyword from set_eng_float_format() (GH395)
• Remove force_unicode keyword from DataFrame.to_string(), DataFrame.to_latex(), and DataFrame.to_html(); these function encode in unicode by default (GH2224, GH2225)
• Remove nanRep keyword from DataFrame.to_csv() and DataFrame.to_string() (GH275)
• Remove unique keyword from HDFStore.select_column() (GH3256)
• Remove inferTimeRule keyword from Timestamp.offset() (GH391)
• Remove name keyword from get_data_yahoo() and get_data_google() (commit b921d1a)
• Remove offset keyword from DatetimeIndex constructor (commit 3136390)
• Remove time_rule from several rolling-moment statistical functions, such as rolling_sum() (GH1042)
• Removed neg - boolean operations on numpy arrays in favor of inv ~, as this is going to be deprecated in numpy 1.9 (GH6960)

Deprecations

• The pivot_table() DataFrame.pivot_table() and crosstab() functions now take arguments index and columns instead of rows and cols. A FutureWarning is raised to alert that the old rows and cols arguments will not be supported in a future release (GH5505)

• The DataFrame.drop_duplicates() and DataFrame.duplicated() methods now take argument subset instead of cols to better align with DataFrame.dropna(). A FutureWarning is raised to alert that the old cols arguments will not be supported in a future release (GH6680)

• The DataFrame.to_csv() and DataFrame.to_excel() functions now takes argument columns instead of cols. A FutureWarning is raised to alert that the old cols arguments will not be supported in a future release (GH6645)

• Indexers will warn FutureWarning when used with a scalar indexer and a non-floating point Index (GH4892, GH6960)

```python
# non-floating point indexes can only be indexed by integers / labels
In [1]: pd.Series(1, np.arange(5))[3.0]
pandas/core/index.py:469: FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
Out[1]: 1

In [2]: pd.Series(1, np.arange(5)).iloc[3.0]
pandas/core/index.py:469: FutureWarning: scalar indexers for index type Int64Index should be integers and not floating point
Out[2]: 1

In [3]: pd.Series(1, np.arange(5)).iloc[3.0:4]
pandas/core/index.py:527: FutureWarning: slice indexers when using iloc should be integers and not floating point
Out[3]:
3  1
   dtype: int64

# these are Float64Indexes, so integer or floating point is acceptable
In [4]: pd.Series(1, np.arange(5.))[3]
Out[4]: 1

In [5]: pd.Series(1, np.arange(5.))[3.0]
Out[6]: 1
```

• Numpy 1.9 compat w.r.t. deprecation warnings (GH6960)

• Panel.shift() now has a function signature that matches DataFrame.shift(). The old positional argument lags has been changed to a keyword argument periods with a default value of 1. A FutureWarning is raised if the old argument lags is used by name. (GH6910)

• The order keyword argument of factorize() will be removed. (GH6926).

• Remove the copy keyword from DataFrame.xs(), Panel.major_xs(), Panel.minor_xs(). A view will be returned if possible, otherwise a copy will be made. Previously the user could think that copy=False would ALWAYS return a view. (GH6894)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- The `parallel_coordinates()` function now takes argument `color` instead of `colors`. A `FutureWarning` is raised to alert that the old `colors` argument will not be supported in a future release. (GH6956)

- The `parallel_coordinates()` and `andrews_curves()` functions now take positional argument `frame` instead of `data`. A `FutureWarning` is raised if the old `data` argument is used by name. (GH6956)

- The support for the 'mysql' flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines (GH6900).

- The following `io.sql` functions have been deprecated: `tquery`, `uquery`, `read_frame`, `frame_query`, `write_frame`.

- The percentile_width keyword argument in `describe()` has been deprecated. Use the `percentiles` keyword instead, which takes a list of percentiles to display. The default output is unchanged.

- The default return type of `boxplot()` will change from a dict to a matplotlib Axes in a future release. You can use the future behavior now by passing `return_type='axes'` to boxplot.

Known issues

- OpenPyXL 2.0.0 breaks backwards compatibility (GH7169)

Enhancements

- DataFrame and Series will create a MultiIndex object if passed a tuples dict, See the docs (GH3323)

```
In [65]: pd.Series({('a', 'b'): 1, ('a', 'a'): 0, 
               ('a', 'c'): 2, ('b', 'a'): 3, ('b', 'b'): 4})
.....:
Out[65]:
   a  b
 a  1
 b  4
Length: 5, dtype: int64

In [66]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2}, 
               ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4}, 
               ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6}, 
               ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8}, 
               ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
.....:
Out[66]:
 a   b
 b   a   c   a   b
 A B  1.0  4.0  5.0  8.0  10.0
 C  2.0  3.0  6.0  7.0  NaN
 D NaN NaN NaN NaN  9.0
[3 rows x 5 columns]
```

- Added the `sym_diff` method to Index (GH5543)

- DataFrame.to_latex now takes a longtable keyword, which if True will return a table in a longtable environment. (GH6617)
- Add option to turn off escaping in `DataFrame.to_latex` (GH6472)

- `pd.read_clipboard` will, if the keyword `sep` is unspecified, try to detect data copied from a spreadsheet and parse accordingly. (GH6223)

- Joining a singly-indexed DataFrame with a MultiIndexed DataFrame (GH3662)

  See the docs. Joining MultiIndex DataFrames on both the left and right is not yet supported ATM.

```python
In [67]: household = pd.DataFrame({'household_id': [1, 2, 3],
    ....:     'male': [0, 1, 0],
    ....:     'wealth': [196087.3, 316478.7, 294750],
    ....: },
    ....: columns=['household_id', 'male', 'wealth'])
    ....: .set_index('household_id')

In [68]: household
Out[68]:
         male       wealth
household_id
1          0       196087.3
2          1        316478.7
3          0         294750.0
[3 rows x 2 columns]

In [69]: portfolio = pd.DataFrame({'household_id': [1, 2, 3, 3, 4],
    ....:     'asset_id': ['nl0000301109',
    ....:                      'nl0000289783',
    ....:                      'gb00b03mlx29',
    ....:                      'gb00b03mlx29',
    ....:                      'lu0197800237',
    ....:                      'nl0000289965',
    ....:                      np.nan],
    ....:     'name': ['ABN Amro',
    ....:                    'Robeco',
    ....:                    'Royal Dutch Shell',
    ....:                    'Royal Dutch Shell',
    ....:                    'AAAB Eastern Europe Equity Fund',
    ....:                    'Postbank BioTech Fonds',
    ....:                    np.nan],
    ....:     'share': [1.0, 0.4, 0.6, 0.15, 0.6, 0.25, 1.0],
    ....: },
    ....: columns=['household_id', 'asset_id', 'name', 'share'])
    ....: .set_index(['household_id', 'asset_id'])

In [70]: portfolio
Out[70]:
        name     share
household_id asset_id
1          ABN Amro    1.00
2          Robeco      0.40
(continues on next page)
pandas: powerful Python data analysis toolkit, Release 1.4.2

In [71]: household.join(portfolio, how='inner')
Out[71]:

<table>
<thead>
<tr>
<th>male</th>
<th>wealth</th>
<th>name</th>
<th>share</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>male wealth</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>household_id</td>
<td>asset_id</td>
</tr>
<tr>
<td>1</td>
<td>nl000301109</td>
<td>0196087.3</td>
<td>ABN Amro Decimal 1</td>
</tr>
<tr>
<td>2</td>
<td>nl000289783</td>
<td>316478.7</td>
<td>Robeco 0.40</td>
</tr>
<tr>
<td>3</td>
<td>gb00b03mlx29</td>
<td>316478.7</td>
<td>Royal Dutch Shell 0.60</td>
</tr>
<tr>
<td>4</td>
<td>gb00b03mlx29</td>
<td>294750.0</td>
<td>Royal Dutch Shell 0.15</td>
</tr>
<tr>
<td>5</td>
<td>lu019780237</td>
<td>294750.0</td>
<td>AAB Eastern Europe Equity Fund 0.60</td>
</tr>
<tr>
<td>6</td>
<td>nl000289965</td>
<td>294750.0</td>
<td>Postbank BioTech Fonds 0.25</td>
</tr>
</tbody>
</table>

[7 rows x 4 columns]

- *quotechar*, *doublequote*, and *escapechar* can now be specified when using *DataFrame.to_csv* (GH5414, GH4528)
- Partially sort by only the specified levels of a MultiIndex with the *sort_remaining* boolean kwarg. (GH3984)
- Added *to_julian_date* to *TimeStamp* and *DatetimeIndex*. The Julian Date is used primarily in astronomy and represents the number of days from noon, January 1, 4713 BC. Because nanoseconds are used to define the time in pandas the actual range of dates that you can use is 1678 AD to 2262 AD. (GH4041)
- *DataFrame.to_stata* will now check data for compatibility with Stata data types and will upcast when needed. When it is not possible to losslessly upcast, a warning is issued (GH6327)
- *DataFrame.to_stata* and *StataWriter* will accept keyword arguments *time_stamp* and *data_label* which allow the time stamp and dataset label to be set when creating a file. (GH6545)
- *pandas.io.gbq* now handles reading unicode strings properly. (GH5940)
- *Holidays Calendars* are now available and can be used with the *CustomBusinessDay* offset (GH6719)
- *Float64Index* is now backed by a *float64* dtype ndarray instead of an *object* dtype array (GH6471).
- Implemented *Panel.pct_change* (GH6904)
- Added *how* option to rolling-moment functions to dictate how to handle resampling; rolling_max() defaults to max, rolling_min() defaults to min, and all others default to mean (GH6297)
- CustomBusinessMonthBegin and CustomBusinessMonthEnd are now available (GH6866)
- *Series.quantile*() and *DataFrame.quantile*() now accept an array of quantiles.
- *describe*() now accepts an array of percentiles to include in the summary statistics (GH4196)
- *pivot_table* can now accept Grouper by index and columns keywords (GH6913)

In [72]: import datetime

In [73]: df = pd.DataFrame({
                        ....:     'Branch': 'A A A A A B'.split(),
                        ....:     'Account': '1234567890'
                        ....:     'Balance': [1000, 2000, 3000, 4000, 5000]}
                        ....:     ['Account', 'Balance'],
                        ....:     index=['Customer 1', 'Customer 2', 'Customer 3', 'Customer 4', 'Customer 5'])
In [74]: df
Out[74]:
<table>
<thead>
<tr>
<th>Branch</th>
<th>Buyer</th>
<th>Quantity</th>
<th>Date</th>
<th>PayDay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Carl</td>
<td>1</td>
<td>2013-11-01 13:00:00</td>
<td>2013-10-04 00:00:00</td>
</tr>
<tr>
<td>A</td>
<td>Mark</td>
<td>3</td>
<td>2013-09-01 13:05:00</td>
<td>2013-10-15 13:05:00</td>
</tr>
<tr>
<td>A</td>
<td>Carl</td>
<td>5</td>
<td>2013-10-01 20:00:00</td>
<td>2013-09-05 20:00:00</td>
</tr>
<tr>
<td>A</td>
<td>Carl</td>
<td>1</td>
<td>2013-10-02 10:00:00</td>
<td>2013-11-02 10:00:00</td>
</tr>
<tr>
<td>A</td>
<td>Joe</td>
<td>8</td>
<td>2013-11-01 20:00:00</td>
<td>2013-10-07 20:00:00</td>
</tr>
<tr>
<td>B</td>
<td>Joe</td>
<td>1</td>
<td>2013-10-02 10:00:00</td>
<td>2013-09-05 10:00:00</td>
</tr>
</tbody>
</table>

[6 rows x 5 columns]

In [75]: df.pivot_table(values='Quantity',
                          index=pd.Grouper(freq='M', key='Date'),
                          columns=pd.Grouper(freq='M', key='PayDay'),
                          aggfunc=np.sum)
Out[75]:
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013-09-30</td>
<td>NaN</td>
<td>3.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-10-31</td>
<td>6.0</td>
<td>NaN</td>
<td>1.0</td>
</tr>
<tr>
<td>2013-11-30</td>
<td>9.0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

[3 rows x 3 columns]

- Arrays of strings can be wrapped to a specified width (str.wrap) (GH6999)
- Add nsmallest() and Series.nlargest() methods to Series, See the docs (GH3960)
- PeriodIndex fully supports partial string indexing like DatetimeIndex (GH7043)
Out[78]:
2013-01-01 09:00  0.015696
2013-01-01 10:00 -2.242685
2013-01-01 11:00  1.150036
2013-01-01 12:00  0.991946
2013-01-01 13:00  0.953324
...  
2013-01-05 08:00  0.285296
2013-01-05 09:00  0.484288
2013-01-05 10:00  1.363482
2013-01-05 11:00 -0.781105
2013-01-05 12:00 -0.468018
Freq: H, Length: 100, dtype: float64

In [79]: ps['2013-01-02']
Out[79]:
2013-01-02 00:00  0.553439
2013-01-02 01:00  1.318152
2013-01-02 02:00 -0.469305
2013-01-02 03:00  0.675554
2013-01-02 04:00 -1.817027
...  
2013-01-02 19:00  0.036142
2013-01-02 20:00 -2.074978
2013-01-02 21:00  0.247792
2013-01-02 22:00 -0.897157
2013-01-02 23:00 -0.136795
Freq: H, Length: 24, dtype: float64

- **read_excel** can now read milliseconds in Excel dates and times with xlrd >= 0.9.3.  
  (GH5945)
- **pd.stats.moments.rolling_var** now uses Welford’s method for increased numerical stability (GH6817)
- **pd.expanding_apply** and **pd.rolling_apply** now take args and kwargs that are passed on to the func (GH6289)
- **DataFrame.rank()** now has a percentage rank option (GH5971)
- **Series.rank()** now has a percentage rank option (GH5971)
- **Series.rank()** and **DataFrame.rank()** now accept `method='dense'` for ranks without gaps (GH6514)
- Support passing encoding with xlwt (GH3710)
- Refactor Block classes removing **Block.items** attributes to avoid duplication in item handling (GH6745, GH6988).
- Testing statements updated to use specialized asserts (GH6175)
**Performance**

- Performance improvement when converting `DatetimeIndex` to floating ordinals using `DatetimeConverter` (GH6636)
- Performance improvement for `DataFrame.shift` (GH5609)
- Performance improvement in indexing into a MultiIndexed Series (GH5567)
- Performance improvements in single-dtyped indexing (GH6484)
- Improve performance of DataFrame construction with certain offsets, by removing faulty caching (e.g. MonthEnd, BusinessMonthEnd), (GH6479)
- Improve performance of `CustomBusinessDay` (GH6584)
- Improve performance of slice indexing on Series with string keys (GH6341, GH6372)
- Performance improvement for `DataFrame.from_records` when reading a specified number of rows from an iterable (GH6700)
- Performance improvements in timedelta conversions for integer dtypes (GH6754)
- Improved performance of compatible pickles (GH6899)
- Improve performance in certain reindexing operations by optimizing `take_2d` (GH6749)
- `GroupBy.count()` is now implemented in Cython and is much faster for large numbers of groups (GH7016).

**Experimental**

There are no experimental changes in 0.14.0

**Bug fixes**

- Bug in Series ValueError when index doesn’t match data (GH6532)
- Prevent segfault due to MultiIndex not being supported in HDFStore table format (GH1848)
- Bug in `pd.DataFrame.sort_index` where mergesort wasn’t stable when ascending=False (GH6399)
- Bug in `pd.tseries.frequencies.to_offset` when argument has leading zeros (GH6391)
- Bug in version string gen. for dev versions with shallow clones / install from tarball (GH6127)
- Inconsistent tz parsing `Timestamp/to_datetime` for current year (GH5958)
- Indexing bugs with reordered indexes (GH6252, GH6254)
- Bug in `.xs` with a Series multiindex (GH6258, GH5684)
- Bug in conversion of a string types to a DatetimeIndex with a specified frequency (GH6273, GH6274)
- Bug in `eval` where type-promotion failed for large expressions (GH6205)
- Bug in interpolate with inplace=True (GH6281)
- HDFStore.remove now handles start and stop (GH6177)
- HDFStore.select_as_multiple handles start and stop the same way as select (GH6177)
- HDFStore.select_as_coordinates and select_column works with a where clause that results in filters (GH6177)
- Regression in join of non_unique_indexes (GH6329)
• Issue with groupby agg with a single function and a mixed-type frame (GH6337)
• Bug in DataFrame.replace() when passing a non-bool to_replace argument (GH6332)
• Raise when trying to align on different levels of a MultiIndex assignment (GH3738)
• Bug in setting complex dtypes via boolean indexing (GH6345)
• Bug in TimeGrouper/resample when presented with a non-monotonic DatetimeIndex that would return invalid results. (GH4161)
• Bug in index name propagation in TimeGrouper/resample (GH4161)
• TimeGrouper has a more compatible API to the rest of the groupers (e.g. groups was missing) (GH3881)
• Bug in multiple grouping with a TimeGrouper depending on target column order (GH6764)
• Bug in pd.eval when parsing strings with possible tokens like '&' (GH6351)
• Bug correctly handle placements of -inf in Panels when dividing by integer 0 (GH6178)
• DataFrame.shift with axis=1 was raising (GH6371)
• Disabled clipboard tests until release time (run locally with nosetests -A disabled) (GH6048).
• Bug in DataFrame.replace() when passing a nested dict that contained keys not in the values to be replaced (GH6342)
• str.match ignored the na flag (GH6609).
• Bug in take with duplicate columns that were not consolidated (GH6240)
• Bug in interpolate changing dtypes (GH6290)
• Bug in Series.get was using a buggy access method (GH6383)
• Bug in hdfstore queries of the form where=[('date', '>=', datetime(2013,1,1)), ('date', '<=', datetime(2014,1,1))] (GH6313)
• Bug in DataFrame.dropna with duplicate indices (GH6355)
• Regression in chained getitem indexing with embedded list-like from 0.12 (GH6394)
• Float64Index with nans not comparing correctly (GH6401)
• eval/query expressions with strings containing the @ character will now work (GH6366).
• Bug in Series.reindex when specifying a method with some nan values was inconsistent (noted on a resample) (GH6418)
• Bug in DataFrame.replace() where nested dicts were erroneously depending on the order of dictionary keys and values (GH5338).
• Performance issue in concatenating with empty objects (GH3259)
• Clarify sorting of sym_diff on Index objects with NaN values (GH6444)
• Regression in MultiIndex.from_product with a DatetimeIndex as input (GH6439)
• Bug in str.extract when passed a non-default index (GH6348)
• Bug in str.split when passed pat=None and n=1 (GH6466)
• Bug in io.data.DataReader when passed "F-F_Momentum_Factor" and data_source="famafrench" (GH6460)
• Bug in sum of a timedelta64[ns] series (GH6462)
• Bug in resample with a timezone and certain offsets (GH6397)
• Bug in iat/iloc with duplicate indices on a Series (GH6493)
• Bug in read_html where nan’s were incorrectly being used to indicate missing values in text. Should use the empty string for consistency with the rest of pandas (GH5129).
• Bug in read_html tests where redirected invalid URLs would make one test fail (GH6445).
• Bug in multi-axis indexing using .loc on non-unique indices (GH6504)
• Bug that caused _ref_locs corruption when slice indexing across columns axis of a DataFrame (GH6525)
• Regression from 0.13 in the treatment of numpy datetime64 non-ns dtypes in Series creation (GH6529)
• .names attribute of MultilnDEXes passed to set_index are now preserved (GH6459).
• Bug in setitem with a duplicate index and an alignable rhs (GH6541)
• Bug in setitem with .loc on mixed integer Indexes (GH6546)
• Bug in pd.read_stata which would use the wrong data types and missing values (GH6327)
• Bug in DataFrame.to_stata that lead to data loss in certain cases, and could be exported using the wrong data types and missing values (GH6335)
• StataWriter replaces missing values in string columns by empty string (GH6802)
• Inconsistent types in Timestamp addition/subtraction (GH6543)
• Bug in preserving frequency across Timestamp addition/subtraction (GH4547)
• Bug in empty list lookup caused IndexError exceptions (GH6536, GH6551)
• Series.quantile raising on an object dtype (GH6555)
• Bug in .xs with a nan in level when dropped (GH6574)
• Bug in fillna with method=’bfill/ffill’ and datetime64[ns] dtype (GH6587)
• Bug in sql writing with mixed dtypes possibly leading to data loss (GH6509)
• Bug in Series.pop (GH6600)
• Bug in iloc indexing when positional indexer matched Int64Index of the corresponding axis and no reordering happened (GH6612)
• Bug in fillna with limit and value specified
• Bug in DataFrame.to_stata when columns have non-string names (GH4558)
• Bug in compat with np.compress, surfaced in (GH6658)
• Bug in binary operations with a rhs of a Series not aligning (GH6681)
• Bug in DataFrame.to_stata which incorrectly handles nan values and ignores with_index keyword argument (GH6685)
• Bug in resample with extra bins when using an evenly divisible frequency (GH4076)
• Bug in consistency of groupby aggregation when passing a custom function (GH6715)
• Bug in resample when how=None resample freq is the same as the axis frequency (GH5955)
• Bug in downcasting inference with empty arrays (GH6733)
• Bug in obj.blocks on sparse containers dropping all but the last items of same for dtype (GH6748)
• Bug in unpickling NaT (NaTType) (GH4606)
• Bug in DataFrame.replace() where regex meta characters were being treated as regex even when regex=False (GH6777).
• Bug in timedelta ops on 32-bit platforms (GH6808)
• Bug in setting a tz-aware index directly via .index (GH6785)
• Bug in expressions.py where numexpr would try to evaluate arithmetic ops (GH6762).
• Bug in Makefile where it didn't remove Cython generated C files with make clean (GH6768)
• Bug in DataFrame._reduce where non bool-like (0/1) integers were being converted into bools. (GH6806)
• Regression from 0.13 with fillna and a Series on datetime-like (GH6344)
• Bug in np.timedelta64 to DatetimeIndex with timezone outputs incorrect results (GH6818)
• Bug in DataFrame.replace() where changing a dtype through replacement would only replace the first occurrence of a value (GH6689)
• Better error message when passing a frequency of ‘MS’ in Period construction (GH5332)
• Bug in Series.__unicode__ when max_rows=None and the Series has more than 1000 rows. (GH6863)
• Bug in groupby.get_group where a datelike wasn’t always accepted (GH5267)
• Bug in groupBy.get_group created by TimeGrouper raises AttributeError (GH6914)
• Bug in DatetimeIndex.tz_localize and DatetimeIndex.tz_convert converting NaT incorrectly (GH5546)
• Bug in arithmetic operations affecting NaT (GH6873)
• Bug in Series.str.extract where the resulting Series from a single group match wasn’t renamed to the group name
• Bug in DataFrame.to_csv where setting index=False ignored the header kwarg (GH6186)
• Bug in DataFrame.plot and Series.plot, where the legend behave inconsistently when plotting to the same axes repeatedly (GH6678)
• Internal tests for patching __finalize__/bug in merge not finalizing (GH6923, GH6927)
• accept TextFileReader in concat, which was affecting a common user idiom (GH6583)
• Bug in C parser with leading white space (GH3374)
• Bug in C parser with delim_whitespace=True and \r-delimited lines
• Bug in python parser with explicit MultiIndex in row following column header (GH6893)
• Bug in Series.rank and DataFrame.rank that caused small floats (<1e-13) to all receive the same rank (GH6886)
• Bug in DataFrame.apply with functions that used *args or **kwargs and returned an empty result (GH6952)
• Bug in sum/mean on 32-bit platforms on overflows (GH6915)
• Moved Panel.shift to NDFrame.slice_shift and fixed to respect multiple dtypes. (GH6959)
• Bug in enabling subplots=True in DataFrame.plot only has single column raises TypeError, and Series.plot raises AttributeError (GH6951)
• Bug in DataFrame.plot draws unnecessary axes when enabling subplots and kind=scatter (GH6951)
• Bug in read_csv from a filesystem with non-utf-8 encoding (GH6807)
• Bug in iloc when setting / aligning (GH6766)
• Bug causing UnicodeEncodeError when get_dummies called with unicode values and a prefix (GH6885)
• Bug in timeseries-with-frequency plot cursor display (GH5453)
• Bug surfaced in groupby.plot when using a Float64Index (GH7025)
• Stopped tests from failing if options data isn’t able to be downloaded from Yahoo (GH7034)
• Bug in parallel_coordinates and radviz where reordering of class column caused possible color/class mismatch (GH6956)
• Bug in radviz and andrews_curves where multiple values of ‘color’ were being passed to plotting method (GH6956)
• Bug in Float64Index.isin() where containing nan s would make indices claim that they contained all the things (GH7066).
• Bug in DataFrame.boxplot where it failed to use the axis passed as the ax argument (GH3578)
• Bug in the XlsxWriter and XlwtWriter implementations that resulted in datetime columns being formatted without the time (GH7075) were being passed to plotting method
• read_fwf() treats None in colspec like regular python slices. It now reads from the beginning or until the end of the line when colspec contains a None (previously raised a TypeError)
• Bug in cache coherence with chained indexing and slicing; add _is_view property to NDFrame to correctly predict views; mark is_copy on xs only if its an actual copy (and not a view) (GH7084)
• Bug in DatetimeIndex creation from string ndarray with dayfirst=True (GH5917)
• Bug in MultiIndex.from_arrays created from DatetimeIndex doesn’t preserve freq and tz (GH7090)
• Bug in unstack raises ValueError when MultiIndex contains PeriodIndex (GH4342)
• Bug in boxplot and hist draws unnecessary axes (GH6769)
• Regression in groupby.nth() for out-of-bounds indexers (GH6621)
• Bug in quantile with datetime values (GH6965)
• Bug in Dataframe.set_index, reindex and pivot don’t preserve DatetimeIndex and PeriodIndex attributes (GH3950, GH5878, GH6631)
• Bug in MultiIndex.get_level_values doesn’t preserve DatetimeIndex and PeriodIndex attributes (GH7092)
• Bug in Groupby doesn’t preserve tz (GH3950)
• Bug in PeriodIndex partial string slicing (GH6716)
• Bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the large_repr set to ‘info’ (GH7105)
• Bug in DatetimeIndex specifying freq raises ValueError when passed value is too short (GH7098)
• Fixed a bug with the info repr not honoring the display.max_info_columns setting (GH6939)
• Bug PeriodIndex string slicing with out of bounds values (GH5407)
• Fixed a memory error in the hashtable implementation/factorizer on resizing of large tables (GH7157)
• Bug in isnull when applied to 0-dimensional object arrays (GH7176)
• Bug in query/eval where global constants were not looked up correctly (GH7178)
• Bug in recognizing out-of-bounds positional list indexers with iloc and a multi-axis tuple indexer (GH7189)
• Bug in setitem with a single value, MultiIndex and integer indices (GH7190, GH7218)
• Bug in expressions evaluation with reversed ops, showing in series-dataframe ops (GH7198, GH7192)
• Bug in multi-axis indexing with > 2 ndim and a MultiIndex (GH7199)
• Fix a bug where invalid eval/query operations would blow the stack (GH5198)

Contributors

A total of 94 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Acanthostega +
• Adam Marcus +
• Alex Gaudio
• Alex Rothberg
• AllenDowney +
• Andrew Rosenfeld +
• Andy Hayden
• Antoine Mazières +
• Benedikt Sauer
• Brad Buran
• Christopher Whelan
• Clark Fitzgerald
• DSM
• Dale Jung
• Dan Allan
• Dan Birken
• Daniel Waeber
• David Jung +
• David Stephens +
• Douglas McNeil
• Garrett Drapala
• Gouthaman Balaraman +
• Guillaume Poulin +
• Jacob Howard +
• Jacob Schaer
• Jason Sexauer +
• Jeff Reback
• Jeff Tratner
• Jeffrey Starr +
• John David Reaver +
• John McNamara
• John W. O’Brien
• Jonathan Chambers
• Joris Van den Bossche
• Julia Evans
• Júlio +
• K.-Michael Aye
• Katie Atkinson +
• Kelsey Jordahl
• Kevin Sheppard +
• Matt Wittmann +
• Matthias Kuhn +
• Max Grender-Jones +
• Michael E. Gruen +
• Mike Kelly
• Nipun Batra +
• Noah Spies +
• PKEuS
• Patrick O’Keefe
• Phillip Cloud
• Pietro Battiston +
• Randy Carnevale +
• Robert Gibboni +
• Skipper Seabold
• SplashDance +
• Stephan Hoyer +
• Tim Cera +
• Tobias Brandt
• Todd Jennings +
• Tom Augspurger
• TomAugspurger
• Yaroslav Halchenko
• agijsberts +
• akittredge
• ankostis +
• anomrake
5.18 Version 0.13

5.18.1 Version 0.13.1 (February 3, 2014)

This is a minor release from 0.13.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:
• Added `infer_datetime_format` keyword to `read_csv/to_datetime` to allow speedups for homogeneously formatted datetimes.
• Will intelligently limit display precision for datetime/timedelta formats.
• Enhanced Panel `apply()` method.
• Suggested tutorials in new Tutorials section.
• Our pandas ecosystem is growing. We now feature related projects in a new Pandas Ecosystem section.
• Much work has been taking place on improving the docs, and a new Contributing section has been added.
• Even though it may only be of interest to devs, we <3 our new CI status page: ScatterCI.

**Warning:** 0.13.1 fixes a bug that was caused by a combination of having numpy < 1.8, and doing chained assignment on a string-like array. Please review the docs, chained indexing can have unexpected results and should generally be avoided.

This would previously segfault:

```
In [1]: df = pd.DataFrame({"A": np.array(['foo', 'bar', 'bah', 'foo', 'bar'])})
In [2]: df["A"].iloc[0] = np.nan
In [3]: df
Out[3]:
   A
0  NaN
1  bar
2  bah
3  foo
4  bar
```

The recommended way to do this type of assignment is:

```
In [4]: df = pd.DataFrame({"A": np.array(['foo', 'bar', 'bah', 'foo', 'bar'])})
In [5]: df.loc[0, "A"] = np.nan
In [6]: df
Out[6]:
   A
0  NaN
1  bar
2  bah
3  foo
4  bar
```
Output formatting enhancements

- df.info() view now display dtype info per column (GH5682)
- df.info() now honors the option max_info_rows, to disable null counts for large frames (GH5974)

```python
In [7]: max_info_rows = pd.get_option("max_info_rows")
In [8]: df = pd.DataFrame(
    ...:     {
    ...:         "A": np.random.randn(10),
    ...:         "B": np.random.randn(10),
    ...:         "C": pd.date_range("20130101", periods=10),
    ...:     }
    ...: )

In [9]: df.iloc[3:6, [0, 2]] = np.nan

# set to not display the null counts
In [10]: pd.set_option("max_info_rows", 0)
In [11]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
    #    Column    Non-Null Count  Dtype
    ---  -------  ---------------  ----
     0   A        7 non-null     float64
     1   B        10 non-null    float64
     2   C        7 non-null     datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 368.0 bytes

# this is the default (same as in 0.13.0)
In [12]: pd.set_option("max_info_rows", max_info_rows)
In [13]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
    #    Column    Non-Null Count    Dtype
    ---  -------  --------------  ----
     0   A        7 non-null       float64
     1   B        10 non-null      float64
     2   C        7 non-null       datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 368.0 bytes

- Add show_dimensions display option for the new DataFrame repr to control whether the dimensions print.

In [14]: df = pd.DataFrame([[1, 2], [3, 4]])
```

(continues on next page)
In [15]: pd.set_option("show_dimensions", False)

In [16]: df
Out[16]:
     0 1 2
0 1 2 3 4

In [17]: pd.set_option("show_dimensions", True)

In [18]: df
Out[18]:
     0 1 2 3 4
0 1 2 3 4
1 1 2 3 4
[2 rows x 2 columns]

• The ArrayFormatter for datetime and timedelta64 now intelligently limit precision based on the values in the array (GH3401)

Previously output might look like:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>today</td>
<td>diff</td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>0 2001-01-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
<td>4491 days, 00:00:00</td>
</tr>
<tr>
<td>1 2004-06-01 00:00:00</td>
<td>2013-04-19 00:00:00</td>
<td>3244 days, 00:00:00</td>
</tr>
</tbody>
</table>

Now the output looks like:

In [19]: df = pd.DataFrame(
               [pd.Timestamp("20010101"), pd.Timestamp("20040601")], columns=["age"]
               )

In [20]: df["today"] = pd.Timestamp("20130419")

In [21]: df["diff"] = df["today"] - df["age"]

In [22]: df
Out[22]:
   age    today    diff
   0 2001-01-01 2013-04-19 4491 days
   1 2004-06-01 2013-04-19 3244 days

[2 rows x 3 columns]
API changes

- Add -NaN and -nan to the default set of NA values (GH5952). See NA Values.

- Added Series.str.get_dummies vectorized string method (GH6021), to extract dummy/indicator variables for separated string columns:

```python
In [23]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])
In [24]: s.str.get_dummies(sep='|')
Out[24]:
     a  b  c
0   1  0  0
1   1  1  0
2   0  0  0
3   1  0  1
[4 rows x 3 columns]
```

- Added the NDFrame.equals() method to compare if two NDFrames are equal have equal axes, dtypes, and values. Added the array_equivalent function to compare if two ndarrays are equal. NaNs in identical locations are treated as equal. (GH5283) See also the docs for a motivating example.

```python
df = pd.DataFrame({'col': ['foo', 0, np.nan]})
df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2, 1, 0])
df.equals(df2)
df.equals(df2.sort_index())
```

- DataFrame.apply will use the reduce argument to determine whether a Series or a DataFrame should be returned when the DataFrame is empty (GH6007).

Previously, calling DataFrame.apply an empty DataFrame would return either a DataFrame if there were no columns, or the function being applied would be called with an empty Series to guess whether a Series or DataFrame should be returned:

```python
In [32]: def applied_func(col):
    ....:     print("Apply function being called with: ", col)
    ....:     return col.sum()
    ....:

In [33]: empty = DataFrame(columns=['a', 'b'])
In [34]: empty.apply(applied_func)
Apply function being called with:  Series([], Length: 0, dtype: float64)
Out[34]:
     a   NaN
     b     NaN
Length: 2, dtype: float64
```

Now, when apply is called on an empty DataFrame: if the reduce argument is True a Series will returned, if it is False a DataFrame will be returned, and if it is None (the default) the function being applied will be called with an empty series to try and guess the return type.

```python
In [35]: empty.apply(applied_func, reduce=True)
Out[35]:
```
a  NaN
b  NaN
Length: 2, dtype: float64

In [36]: empty.apply(applied_func, reduce=False)
Out[36]:
Empty DataFrame
Columns: [a, b]
Index: []
[0 rows x 2 columns]

Prior version deprecations/changes

There are no announced changes in 0.13 or prior that are taking effect as of 0.13.1

Deprecations

There are no deprecations of prior behavior in 0.13.1

Enhancements

- `pd.read_csv` and `pd.to_datetime` learned a new `infer_datetime_format` keyword which greatly improves parsing perf in many cases. Thanks to @exual for suggesting and @danbirken for rapidly implementing. (GH5490, GH6021)

  If `parse_dates` is enabled and this flag is set, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

  ```python
  # Try to infer the format for the index column
  df = pd.read_csv(
      "foo.csv", index_col=0, parse_dates=True, infer_datetime_format=True
  )
  ```

- `date_format` and `datetime_format` keywords can now be specified when writing to excel files (GH4133)
- `MultiIndex.from_product` convenience function for creating a MultiIndex from the cartesian product of a set of iterables (GH6055):

  In [25]: shades = ["light", "dark"]
  
  In [26]: colors = ["red", "green", "blue"]
  
  In [27]: pd.MultiIndex.from_product([shades, colors], names=["shade", "color"])
  Out[27]:
  MultiIndex([('light', 'red'), ('light', 'green'), ('light', 'blue'), ('dark', 'red'), ('dark', 'green'),] (continues on next page)
• Panel `apply()` will work on non-ufuncs. See [the docs](#).

```python
In [28]: import pandas._testing as tm
In [29]: panel = tm.makePanel(5)
In [30]: panel
Out[30]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D
In [31]: panel['ItemA']
```

```
  ItemA  ItemB  ItemC  ItemD
2000-01-03 -0.67369  0.57704  -1.34431  -1.46938
2000-01-04  0.11365 -1.71500   0.84488   0.35702
2000-01-05 -1.47843  1.07577  -0.67460
2000-01-06 -0.37065  0.84488  -0.10905  -1.77690
2000-01-07  1.64356 -0.96891
```

Specifying an `apply` that operates on a `Series` (to return a single element)

```python
In [32]: panel.apply(lambda x: x.dtype, axis='items')
```

```
  ItemA  ItemB  ItemC  ItemD
2000-01-03 float64 float64 float64 float64
2000-01-04 float64 float64 float64 float64
2000-01-05 float64 float64 float64 float64
2000-01-06 float64 float64 float64 float64
2000-01-07 float64 float64 float64 float64
```

A similar reduction type operation

```python
In [33]: panel.apply(lambda x: x.sum(), axis='major_axis')
```

```
  ItemA  ItemB  ItemC
A     -1.10877  1.09011 2.984435
B     -3.70576   0.40920 1.866240
C     2.11086  2.96050  0.974967
D    -4.53279   0.30320  3.685193
```

This is equivalent to

```
In [34]: panel.sum('major_axis')
```

```
Out[34]:
    ItemA  ItemB  ItemC
A -1.108775 -1.090118 -2.984435
B -3.705764  0.409204  1.866240
C  2.110856  2.960500 -0.974967
D -4.532785  0.303202 -3.685193
```

[4 rows x 3 columns]

A transformation operation that returns a Panel, but is computing the z-score across the major_axis

```
In [35]: result = panel.apply(lambda x: (x - x.mean()) / x.std(),
                        axis='major_axis')
```

```
In [36]: result
```

```
Out[36]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D
```

```
In [37]: result['ItemA']
```

```
Out[37]:
          A         B         C         D
2000-01-03 -0.535778  1.500802 -1.506416 -0.681456
2000-01-04  0.397628 -1.108752  0.360481  1.529895
2000-01-05 -1.489811 -0.339412  0.557374  0.280845
2000-01-06  0.885279  0.421830 -0.453013 -1.053785
2000-01-07  0.742682 -0.474468  1.041575 -0.075499
```

[5 rows x 4 columns]

- Panel `apply()` operating on cross-sectional slabs. (GH1148)

```
In [38]: def f(x):
    ....:     return ((x.T - x.mean(1)) / x.std(1)).T
    ....:

In [39]: result = panel.apply(f, axis=['items', 'major_axis'])
```

```
In [40]: result
```

```
Out[40]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC
```

```
In [41]: result.loc[:, :, 'ItemA']
```

(continues on next page)
This is equivalent to the following

```
In [42]: result = pd.Panel({ax: f(panel.loc[:, :, ax]) for ax in panel.minor_axis})
In [43]: result
Out[43]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC
```

```
In [44]: result.loc[:, :, 'ItemA']
Out[44]:
A     B     C     D
2000-01-03  0.012922 -0.030874 -0.629546 -0.757034
2000-01-04  0.392053 -1.071665  0.163228  0.548188
2000-01-05 -1.093650 -0.640898  0.385734 -1.154310
2000-01-06  1.005446 -1.154593 -0.595615 -0.809185
2000-01-07  0.783051 -0.198053  0.919339 -1.052721
[5 rows x 4 columns]
```

**Performance**

Performance improvements for 0.13.1

- Series datetime/timedelta binary operations ([GH5801](#))
- DataFrame count/dropna for axis=1
- Series.str.contains now has a `regex=False` keyword which can be faster for plain (non-regex) string patterns. ([GH5879](#))
- Series.str.extract ([GH5944](#))
- dtypes/ftypes methods ([GH5968](#))
- indexing with object dtypes ([GH5968](#))
- DataFrame.apply ([GH6013](#))
- Regression in JSON IO ([GH5765](#))
- Index construction from Series ([GH6150](#))
Experimental

There are no experimental changes in 0.13.1

Bug fixes

- Bug in `io.wb.get_countries` not including all countries (GH6008)
- Bug in Series replace with timestamp dict (GH5797)
- `read_csv/read_table` now respects the `prefix` kwarg (GH5732).
- Bug in selection with missing values via `.ix` from a duplicate indexed DataFrame failing (GH5835)
- Fix issue of boolean comparison on empty DataFrames (GH5808)
- Bug in `isnull` handling `NaT` in an object array (GH5443)
- Bug in `to_datetime` when passed a np.nan or integer datelike and a format string (GH5863)
- Bug in `groupby` dtype conversion with datetimelike (GH5869)
- Regression in handling of empty Series as indexers to Series (GH5877)
- Bug in internal caching, related to (GH5727)
- Testing bug in reading JSON/msgpack from a non-filepath on windows under py3 (GH5874)
- Bug when assigning to `.ix[tuple(...)]]` (GH5896)
- Bug in fully reindexing a Panel (GH5905)
- Bug in `idxmin/max` with object dtypes (GH5914)
- Bug in `BusinessDay` when adding n days to a date not on offset when n>5 and n%5==0 (GH5890)
- Bug in assigning to chained series with a series via `ix` (GH5928)
- Bug in creating an empty DataFrame, copying, then assigning (GH5932)
- Bug in Dataframe.tail with empty frame (GH5846)
- Bug in propagating metadata on `resample` (GH5862)
- Fixed string-representation of `NaT` to be “NaT” (GH5708)
- Fixed string-representation for Timestamp to show nanoseconds if present (GH5912)
- `pd.match` not returning passed sentinel
- `Panel.to_frame()` no longer fails when `major_axis` is a `MultiIndex` (GH5402).
- Bug in `pd.read_msgpack` with inferring a `DateTimeIndex` frequency incorrectly (GH5947)
- Fixed `to_datetime` for array with both Tz-aware datetimes and `NaT`’s (GH5961)
- Bug in rolling skew/kurtosis when passed a Series with bad data (GH5749)
- Bug in `scipy.interpolate` methods with a datetime index (GH5975)
- Bug in `NaT` comparison if a mixed datetime/np.datetime64 with `NaT` were passed (GH5968)
- Fixed bug with `pd.concat` losing dtype information if all inputs are empty (GH5742)
- Recent changes in IPython cause warnings to be emitted when using previous versions of pandas in QTConsole, now fixed. If you're using an older version and need to suppress the warnings, see (GH5922).
- Bug in merging `timedelta` dtypes (GH5695)
pandas: powerful Python data analysis toolkit, Release 1.4.2

- Bug in plotting.scatter_matrix function. Wrong alignment among diagonal and off-diagonal plots, see (GH5497).
- Regression in Series with a MultiIndex via ix (GH6018)
- Bug in Series.xs with a MultiIndex (GH6018)
- Bug in Series construction of mixed type with datelike and an integer (which should result in object type and not automatic conversion) (GH6028)
- Possible segfault when chained indexing with an object array under NumPy 1.7.1 (GH6026, GH6056)
- Bug in setting using fancy indexing a single element with a non-scalar (e.g. a list), (GH6043)
- to_sql did not respect if_exists (GH4110 GH4304)
- Regression in .get(None) indexing from 0.12 (GH5652)
- Subtle iloc indexing bug, surfaced in (GH6059)
- Bug with insert of strings into DatetimeIndex (GH5818)
- Fixed unicode bug in to_html/HTML repr (GH6098)
- Fixed missing arg validation in get_options_data (GH6105)
- Bug in assignment with duplicate columns in a frame where the locations are a slice (e.g. next to each other) (GH6120)
- Bug in propagating _ref_locs during construction of a DataFrame with dups index/columns (GH6121)
- Bug in DataFrame.apply when using mixed datelike reductions (GH6125)
- Bug in DataFrame.append when appending a row with different columns (GH6129)
- Bug in DataFrame construction with recarray and non-ns datetime dtype (GH6140)
- Bug in .loc setitem indexing with a dataframe on rhs, multiple item setting, and a datetimelike (GH6152)
- Fixed a bug in query/eval during lexicographic string comparisons (GH6155).
- Fixed a bug in query where the index of a single-element Series was being thrown away (GH6148).
- Bug in HDFStore on appending a dataframe with MultiIndexed columns to an existing table (GH6167)
- Consistency with dtypes in setting an empty DataFrame (GH6171)
- Bug in selecting on a MultiIndex HDFStore even in the presence of under specified column spec (GH6169)
- Bug in nanops.var with ddof=1 and 1 elements would sometimes return inf rather than nan on some platforms (GH6136)
- Bug in Series and DataFrame bar plots ignoring the use_index keyword (GH6209)
- Bug in groupby with mixed str/int under python3 fixed; argsort was failing (GH6212)

Contributors

A total of 52 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Alex Rothberg
- Alok Singhal +
- Andrew Burrows +
- Andy Hayden
• Bjorn Arneson +
• Brad Buran
• Caleb Epstein
• Chapman Siu
• Chase Albert +
• Clark Fitzgerald +
• DSM
• Dan Birken
• Daniel Waeber +
• David Wolever +
• Doran Deluz +
• Douglas McNeil +
• Douglas Rudd +
• Drazen Lucanin
• Elliot S +
• Felix Lawrence +
• George Kuan +
• Guillaume Gay +
• Jacob Schaer
• Jan Wagner +
• Jeff Tratner
• John McNamara
• Joris Van den Bossche
• Julia Evans +
• Kieran O’Mahony
• Michael Schatzow +
• Naveen Michaud-Agrawal +
• Patrick O’Keeffe +
• Phillip Cloud
• Roman Pekar
• Skipper Seabold
• Spencer Lyon
• Tom Augspurger +
• TomAugspurger
• acorbe +
• akittredge +

5.18. Version 0.13 3617
5.18.2 Version 0.13.0 (January 3, 2014)

This is a major release from 0.12.0 and includes a number of API changes, several new features and enhancements along with a large number of bug fixes.

Highlights include:

- support for a new index type Float64Index, and other Indexing enhancements
- HDFStore has a new string based syntax for query specification
- support for new methods of interpolation
- updated timedelta operations
- a new string manipulation method extract
- Nanosecond support for Offsets
- isin for DataFrames

Several experimental features are added, including:

- new eval/query methods for expression evaluation
- support for msgpack serialization
- an i/o interface to Google’s BigQuery

There are several new or updated docs sections including:

- Comparison with SQL, which should be useful for those familiar with SQL but still learning pandas.
- Comparison with R, idiom translations from R to pandas.
- Enhancing Performance, ways to enhance pandas performance with eval/query.

**Warning:** In 0.13.0 Series has internally been refactored to no longer sub-class ndarray but instead subclass NDFrame, similar to the rest of the pandas containers. This should be a transparent change with only very limited API implications. See Internal Refactoring
API changes

- **read_excel** now supports an integer in its `sheetname` argument giving the index of the sheet to read in (GH4301).
- Text parser now treats anything that reads like inf (“inf”, “Inf”, “-Inf”, “iNf”, etc.) as infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
- **pandas** now is Python 2/3 compatible without the need for `2to3` thanks to @jtratner. As a result, pandas now uses iterators more extensively. This also led to the introduction of substantive parts of the Benjamin Peterson’s six library into compat. (GH4384, GH4375, GH4372)
- `pandas.util.compat` and `pandas.util.py3compat` have been merged into `pandas.compat`. `pandas.compat` now includes many functions allowing 2/3 compatibility. It contains both list and iterator versions of range, filter, map and zip, plus other necessary elements for Python 3 compatibility. `lmap`, `lzip`, `lrange` and `lfilter` all produce lists instead of iterators, for compatibility with numpy, subscripting and pandas constructors. (GH4384, GH4375, GH4372)
- **Series.get** with negative indexers now returns the same as `[]` (GH4390)
- Changes to how `Index` and `MultiIndex` handle metadata (levels, labels, and names) (GH4039):

```python
# previously, you would have set levels or labels directly
>>> pd.index.levels = [[1, 2, 3, 4], [1, 2, 4, 4]]

# now, you use the set_levels or set_labels methods
>>> index = pd.index.set_levels([[1, 2, 3, 4], [1, 2, 4, 4]])

# similarly, for names, you can rename the object
# but setting names is not deprecated
>>> index = pd.index.set_names(['bob', 'cranberry'])

# and all methods take an inplace kwarg - but return None
>>> pd.index.set_names(['bob', 'cranberry'], inplace=True)
```

- All division with `NDFrame` objects is now `true_division`, regardless of the future import. This means that operating on pandas objects will by default use floating point division, and return a floating point dtype. You can use `//` and `floordiv` to do integer division.

Integer division

```python
In [3]: arr = np.array([1, 2, 3, 4])

In [4]: arr2 = np.array([5, 3, 2, 1])

In [5]: arr / arr2
Out[5]: array([0, 0, 1, 4])

In [6]: pd.Series(arr) // pd.Series(arr2)
Out[6]:
0  0
1  0
2  1
3  4
dtype: int64
```

True Division
```
In [7]: pd.Series(arr) / pd.Series(arr2)  # no future import required
Out[7]:
   0    0.200000
   1    0.666667
   2    1.500000
   3    4.000000
dtype: float64
```

- Infer and downcast dtype if downcast='infer' is passed to fillna/ffill/bfill (GH4604)
- `__nonzero__` for all NDFrame objects, will now raise a `ValueError`, this reverts back to (GH1073, GH4633) behavior. See gotchas for a more detailed discussion.

This prevents doing boolean comparison on `entire` pandas objects, which is inherently ambiguous. These all will raise a `ValueError`.

```
>>> df = pd.DataFrame({'A': np.random.randn(10),
...                     'B': np.random.randn(10),
...                     'C': pd.date_range('20130101', periods=10)
...                     })

>>> if df:
...     pass
...
Traceback (most recent call last):
  ...
ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

```
>>> df1 = df
>>> df2 = df
>>> df1 and df2
Traceback (most recent call last):
  ...
ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

```
>>> d = [1, 2, 3]
>>> s1 = pd.Series(d)
>>> s2 = pd.Series(d)
>>> s1 and s2
Traceback (most recent call last):
  ...
ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

Added the `.bool()` method to NDFrame objects to facilitate evaluating of single-element boolean Series:

```
In [1]: pd.Series([True]).bool()
Out[1]: True

In [2]: pd.Series([False]).bool()
Out[2]: False
```

(continues on next page)
In [3]: pd.DataFrame([[True]]).bool()
Out[3]: True

In [4]: pd.DataFrame([[False]]).bool()
Out[4]: False

• All non-Index NDFrames (Series, DataFrame, Panel, Panel4D, SparsePanel, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (add, sub, mul, etc.). SparsePanel does not support pow or mod with non-scalars. (GH3765)

• Series and DataFrame now have a mode() method to calculate the statistical mode(s) by axis/Series. (GH5367)

• Chained assignment will now by default warn if the user is assigning to a copy. This can be changed with the option mode.chained_assignment, allowed options are raise/warn/None. See the docs.

In [5]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [6]: pd.set_option('chained_assignment', 'warn')

The following warning / exception will show if this is attempted.

In [7]: dfc.loc[0]['A'] = 1111

Traceback (most recent call last)
...
SettingWithCopyWarning:
  A value is trying to be set on a copy of a slice from a DataFrame. Try using .loc[row_index,col_indexer] = value instead

Here is the correct method of assignment.

In [8]: dfc.loc[0, 'A'] = 11

In [9]: dfc
Out[9]:
        A  B
       0   11  1
       1  bbb  2
       2  ccc  3

• Panel.reindex has the following call signature Panel.reindex(items=None, major_axis=None, minor_axis=None, **kwargs) to conform with other NDFrame objects. See Internal Refactoring for more information.

• Series.argmin and Series.argmax are now aliased to Series.idxmin and Series.idxmax. These return the index of the min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element. (GH6214)
Prior version deprecations/changes

These were announced changes in 0.12 or prior that are taking effect as of 0.13.0

- Remove deprecated Factor (GH3650)
- Remove deprecated set_printoptions/reset_printoptions (GH3046)
- Remove deprecated _verbose_info (GH3215)
- Remove deprecated read_clipboard/to_clipboard/ExcelFile/ExcelWriter from pandas.io.parsers (GH3717) These are available as functions in the main pandas namespace (e.g. pd.read_clipboard)
- default for tupleize_cols is now False for both to_csv and read_csv. Fair warning in 0.12 (GH3604)
- default for display.max_seq_len is now 100 rather than None. This activates truncated display ("...") of long sequences in various places. (GH3391)

Deprecations

Deprecated in 0.13.0

- deprecated iterkv, which will be removed in a future release (this was an alias of iteritems used to bypass 2to3’s changes). (GH4384, GH4375, GH4372)
- deprecated the string method match, whose role is now performed more idiomatically by extract. In a future release, the default behavior of match will change to become analogous to contains, which returns a boolean indexer. (Their distinction is strictness: match relies on re.match while contains relies on re.search.) In this release, the deprecated behavior is the default, but the new behavior is available through the keyword argument as_indexer=True.

Indexing API changes

Prior to 0.13, it was impossible to use a label indexer (.loc/.ix) to set a value that was not contained in the index of a particular axis. (GH2578). See the docs

In the Series case this is effectively an appending operation

```
In [10]: s = pd.Series([1, 2, 3])

In [11]: s
Out[11]:
0   1
1   2
2   3
dtype: int64


In [13]: s
Out[13]:
0   1.0
1   2.0
2   3.0
5   5.0
dtype: float64
```
In [14]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
   columns=['A', 'B'])

This would previously KeyError

In [16]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

This is like an append operation.

In [18]: dfi.loc[3] = 5

A Panel setting operation on an arbitrary axis aligns the input to the Panel

In [20]: p = pd.Panel(np.arange(16).reshape(2, 4, 2),
   items=['Item1', 'Item2'],
   major_axis=pd.date_range('2001/1/12', periods=4),
   minor_axis=['A', 'B'], dtype='float64')

In [22]: p.loc[:, :, 'C'] = pd.Series([30, 32], index=p.items)

In [23]: p
Out[23]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to C

In [24]: p.loc[:, :, 'C']
Out[24]:

<table>
<thead>
<tr>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item1</td>
<td>Item2</td>
</tr>
<tr>
<td>2001-01-12</td>
<td>30.0</td>
</tr>
<tr>
<td>2001-01-13</td>
<td>30.0</td>
</tr>
<tr>
<td>2001-01-14</td>
<td>30.0</td>
</tr>
<tr>
<td>2001-01-15</td>
<td>30.0</td>
</tr>
</tbody>
</table>

**Float64Index API change**

- Added a new index type, `Float64Index`. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes [], .ix, .loc for scalar indexing and slicing work exactly the same. See the docs (GH263)

  Construction is by default for floating type values.

In [20]: index = pd.Index([1.5, 2, 3, 4.5, 5])

In [21]: index
Out[21]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [22]: s = pd.Series(range(5), index=index)

In [23]: s
Out[23]:

| 1.5 | 0 |
| 2.0 | 1 |
| 3.0 | 2 |
| 4.5 | 3 |
| 5.0 | 4 |

dtype: int64

Scalar selection for [], .ix, .loc will always be label based. An integer will match an equal float index (e.g. 3 is equivalent to 3.0)

In [24]: s[3]
Out[24]: 2

In [25]: s.loc[3]
Out[25]: 2

The only positional indexing is via iloc

In [26]: s.iloc[3]
Out[26]: 3
A scalar index that is not found will raise **KeyError**

Slicing is **ALWAYS** on the values of the index, for [], ix, loc and **ALWAYS** positional with iloc

```
In [27]: s[2:4]
Out[27]:
2.0  1
3.0  2
dtype: int64

In [28]: s.loc[2:4]
Out[28]:
2.0  1
3.0  2
dtype: int64

In [29]: s.iloc[2:4]
Out[29]:
3.0  2
4.5  3
dtype: int64
```

In float indexes, slicing using floats are allowed

```
In [30]: s[2.1:4.6]
Out[30]:
3.0  2
4.5  3
dtype: int64

In [31]: s.loc[2.1:4.6]
Out[31]:
3.0  2
4.5  3
dtype: int64
```

- Indexing on other index types are preserved (and positional fallback for [], ix), with the exception, that floating point slicing on indexes on non `Float64Index` will now raise a **TypeError**.

```
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type →(Int64Index)
```

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

```
In [3]: pd.Series(range(5))[3.0]
Out[3]: 3
```
HDFStore API changes

- Query Format Changes. A much more string-like query format is now supported. See the docs.

```python
In [32]: path = 'test.h5'
In [33]: dfq = pd.DataFrame(np.random.randn(10, 4),
                           columns=list('ABCD'),
                           index=pd.date_range('20130101', periods=10))
In [34]: dfq.to_hdf(path, 'dfq', format='table', data_columns=True)

Use boolean expressions, with in-line function evaluation.

```python
In [35]: pd.read_hdf(path, 'dfq',
                           where="index>Timestamp('20130104') & columns=['A', 'B']")
Out[35]:
       A     B
2013-01-05 -0.424972  0.567020
2013-01-06  0.673690  0.113648
2013-01-07  0.404705  0.577046
2013-01-08 -0.370647 -1.157892
2013-01-09  1.075770 -0.109050
2013-01-10  0.357021 -0.674600

Use an inline column reference

```python
In [36]: pd.read_hdf(path, 'dfq',
                           where="A>0 or C>0")
Out[36]:
       A     B     C     D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-07  0.404705  0.577046 -1.715002 -1.039268
2013-01-09  1.075770 -0.109050  1.643563 -1.469388
2013-01-10  0.357021 -0.674600 -1.776904  0.968914
```

- the format keyword now replaces the table keyword; allowed values are fixed(f) or table(t) the same defaults as prior < 0.13.0 remain, e.g. put implies fixed format and append implies table format. This default format can be set as an option by setting io.hdf.default_format.
In [41]: df.to_hdf(path, 'df_fixed')

In [42]: with pd.HDFStore(path) as store:
   ....:     print(store)
   ....:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

- Significant table writing performance improvements
- handle a passed Series in table format (GH4330)
- can now serialize a timedelta64[ns] dtype in a table (GH3577), See the docs.
- added an is_open property to indicate if the underlying file handle is_open; a closed store will now report ‘CLOSED’ when viewing the store (rather than raising an error) (GH4409)
- a close of a HDFStore now will close that instance of the HDFStore but will only close the actual file if the ref count (by PyTables) w.r.t. all of the open handles are 0. Essentially you have a local instance of HDFStore referenced by a variable. Once you close it, it will report closed. Other references (to the same file) will continue to operate until they themselves are closed. Performing an action on a closed file will raise ClosedFileError

In [43]: path = 'test.h5'

In [44]: df = pd.DataFrame(np.random.randn(10, 2))

In [45]: store1 = pd.HDFStore(path)

In [46]: store2 = pd.HDFStore(path)

In [47]: store1.append('df1', df)

In [48]: store2.append('df2', df)

In [49]: store1
Out[49]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [50]: store2
Out[50]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [51]: store1.close()

In [52]: store2
Out[52]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [53]: store2.close()

In [54]: store2
• removed the _quiet attribute, replace by a DuplicateWarning if retrieving duplicate rows from a table (GH4367)

• removed the warn argument from open. Instead a PossibleDataLossError exception will be raised if you try to use mode='w' with an OPEN file handle (GH4367)

• allow a passed locations array or mask as a where condition (GH4467). See the docs for an example.

• add the keyword dropna=True to append to change whether ALL nan rows are not written to the store (default is True, ALL nan rows are NOT written), also settable via the option io.hdf.dropna_table (GH4625)

• pass through store creation arguments; can be used to support in-memory stores

DataFrame repr changes

The HTML and plain text representations of DataFrame now show a truncated view of the table once it exceeds a certain size, rather than switching to the short info view (GH4886, GH5550). This makes the representation more consistent as small DataFrames get larger.

| 2010-03-30 | 13.55 | 13.64 | 13.18 | 13.28 | 142055200 | 12.70 |

771 rows x 6 columns

To get the info view, call DataFrame.info(). If you prefer the info view as the repr for large DataFrames, you can set this by running set_option('display.large_repr', 'info').

Enhancements

• df.to_clipboard() learned a new excel keyword that let’s you paste df data directly into excel (enabled by default). (GH5070).

• read_html now raises a URLError instead of catching and raising a ValueError (GH4303, GH4305)

• Added a test for read_clipboard() and to_clipboard() (GH4282)

• Clipboard functionality now works with PySide (GH4282)

• Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)

• to_dict now takes records as a possible out type. Returns an array of column-keyed dictionaries. (GH4936)

• NaN handing in get_dummies (GH4446) with dummy_na

# previously, nan was erroneously counted as 2 here
# now it is not counted at all
In [55]: pd.get_dummies([1, 2, np.nan])
Out[55]:

(continues on next page)
# unless requested

```python
In [56]: pd.get_dummies([1, 2, np.nan], dummy_na=True)
Out[56]:
   1.0  2.0  NaN
      0   1   0
      1   0   1
      2   0   0
```

- `timedelta64[ns]` operations. See the docs.

```
Warning: Most of these operations require numpy >= 1.7
```

Using the new top-level `pd.to_timedelta`, you can convert a scalar or array from the standard timedelta format (produced by `pd.to_csv`) into a timedelta type (`np.timedelta64` in nanoseconds).

```
In [57]: pd.to_timedelta('1 days 06:05:01.00003')
Out[57]: Timedelta('1 days 06:05:01.000030')

In [58]: pd.to_timedelta('15.5us')
Out[58]: Timedelta('0 days 00:00:00.000015500')

In [59]: pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', np.nan])
Out[59]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015500', NaT], dtype='timedelta64[ns]', freq=None)

In [60]: pd.to_timedelta(np.arange(5), unit='s')
Out[60]: TimedeltaIndex(['0 days 00:00:00', '1 days 00:00:01', '2 days 00:00:02', '3 days 00:00:03', '4 days 00:00:04'], dtype='timedelta64[ns]', freq=None)

In [61]: pd.to_timedelta(np.arange(5), unit='d')
Out[61]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype='timedelta64[ns]', freq=None)
```

A Series of dtype `timedelta64[ns]` can now be divided by another `timedelta64[ns]` object, or astyped to yield a `float64` dtyped Series. This is frequency conversion. See the docs for the docs.

```
In [62]: import datetime

In [63]: td = pd.Series(pd.date_range('20130101', periods=4)) - pd.Series(pd.date_range('20121201', periods=4))

In [64]: td[2] += np.timedelta64(datetime.timedelta(minutes=5, seconds=3))
```

(continues on next page)
In [65]: td[3] = np.nan

In [66]: td
Out[66]:
0    31 days 00:00:00
1    31 days 00:00:00
2    31 days 00:05:03
3        NaT
dtype: timedelta64[ns]

# to days
In [67]: td / np.timedelta64(1, 'D')
Out[67]:
0    31.000000
1    31.000000
2    31.003507
3        NaN
dtype: float64

In [68]: td.astype('timedelta64[D]')
Out[68]:
0    31.0
1    31.0
2    31.0
3        NaN
dtype: float64

# to seconds
In [69]: td / np.timedelta64(1, 's')
Out[69]:
0  2678400.0
1  2678400.0
2  2678703.0
3        NaN
dtype: float64

In [70]: td.astype('timedelta64[s]')
Out[70]:
0  2678400.0
1  2678400.0
2  2678703.0
3        NaN
dtype: float64

Dividing or multiplying a timedelta64[ns] Series by an integer or integer Series

In [71]: td * -1
Out[71]:
0   -31 days +00:00:00
1   -31 days +00:00:00
2   -32 days +23:54:57
3        NaT
(continues on next page)
dtype: timedelta64[ns]

In [72]: td * pd.Series([1, 2, 3, 4])
Out[72]:
0 31 days 00:00:00
1 62 days 00:00:00
2 93 days 00:15:09
3 NaT
dtype: timedelta64[ns]

Absolute DateOffset objects can act equivalently to timedeltas

In [73]: from pandas import offsets
In [74]: td + offsets.Minute(5) + offsets.Milli(5)
Out[74]:
0 31 days 00:05:00.005000
1 31 days 00:05:00.005000
2 31 days 00:10:03.005000
3 NaT
dtype: timedelta64[ns]

Fillna is now supported for timedeltas

In [75]: td.fillna(pd.Timedelta(0))
Out[75]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 0 days 00:00:00
dtype: timedelta64[ns]

In [76]: td.fillna(datetime.timedelta(days=1, seconds=5))
Out[76]:
0 31 days 00:00:00
1 31 days 00:00:00
2 31 days 00:05:03
3 1 days 00:00:05
dtype: timedelta64[ns]

You can do numeric reduction operations on timedeltas.

In [77]: td.mean()
Out[77]: Timedelta('31 days 00:01:41')

In [78]: td.quantile(.1)
Out[78]: Timedelta('31 days 00:00:00')

- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to `scipy.stats.gaussian_kde()` (for scipy >= 0.11.0) to set the bandwidth, and to `gkde.evaluate()` to specify the indices at which it is evaluated, respectively. See scipy docs. (GH4298)
- DataFrame constructor now accepts a numpy masked record array (GH3478)
The new vectorized string method `extract` return regular expression matches more conveniently.

```python
In [79]: pd.Series(['a1', 'b2', 'c3']).str.extract('^[ab](\d)$')
Out[79]:
0   1
1   2
2  NaN
Name: 0, dtype: int64
```

Elements that do not match return `NaN`. Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```python
In [80]: pd.Series(['a1', 'b2', 'c3']).str.extract('^[ab](\d)\d$')
Out[80]:
   0  1
0   a 1
1   b 2
2  NaN NaN
Name: 0, dtype: object
```

Elements that do not match return a row of `NaN`. Thus, a Series of messy strings can be converted into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects.

Named groups like

```python
In [81]: pd.Series(['a1', 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)')
```

and optional groups can also be used.

```python
In [82]: pd.Series(['a1', 'b2', '3']).str.extract('(?P<letter>[ab])(?P<digit>\d)')
```

- `read_stata` now accepts Stata 13 format (GH4291)
- `read_fwf` now infers the column specifications from the first 100 rows of the file if the data has correctly separated and properly aligned columns using the delimiter provided to the function (GH4488).
- support for nanosecond times as an offset

**Warning:** These operations require `numpy` >= 1.7
Period conversions in the range of seconds and below were reworked and extended up to nanoseconds. Periods in the nanosecond range are now available.

```
In [83]: pd.date_range('2013-01-01', periods=5, freq='5N')
Out[83]:
DatetimeIndex(['2013-01-01 00:00:00',
               '2013-01-01 00:00:00.000000005',
               '2013-01-01 00:00:00.000000010',
               '2013-01-01 00:00:00.000000015',
               '2013-01-01 00:00:00.000000020'],
dtype='datetime64[ns]', freq='5N')
```

or with frequency as offset

```
In [84]: pd.date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
Out[84]:
DatetimeIndex(['2013-01-01 00:00:00',
               '2013-01-01 00:00:00.000000005',
               '2013-01-01 00:00:00.000000010',
               '2013-01-01 00:00:00.000000015',
               '2013-01-01 00:00:00.000000020'],
dtype='datetime64[ns]', freq='5N')
```

Timestamps can be modified in the nanosecond range

```
In [85]: t = pd.Timestamp('20130101 09:01:02')
In [86]: t + pd.tseries.offsets.Nano(123)
Out[86]: Timestamp('2013-01-01 09:01:02.000000123')
```

- A new method, isin for DataFrames, which plays nicely with boolean indexing. The argument to isin, what we’re comparing the DataFrame to, can be a DataFrame, Series, dict, or array of values. See the docs for more.

To get the rows where any of the conditions are met:

```
In [87]: dfi = pd.DataFrame({
                      'A': [1, 2, 3, 4],
                      'B': ['a', 'b', 'f', 'n']})

In [88]: dfi
Out[88]:
   A  B
0  a  b
1  f  n

In [89]: other = pd.DataFrame({
                         'A': [1, 3, 3, 7],
                         'B': ['e', 'f', 'f', 'e']})

In [90]: mask = dfi.isin(other)

In [91]: mask
Out[91]:
   A  B
0 True False
1 False False
2 True True
```

(continues on next page)
• Series now supports a to_frame method to convert it to a single-column DataFrame (GH5164)

• All R datasets listed here http://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html can now be loaded into pandas objects

```python
# note that pandas.rpy was deprecated in v0.16.0
import pandas.rpy.common as com
com.load_data('Titanic')
```

• tz_localize can infer a fall daylight savings transition based on the structure of the unlocalized data (GH4230), see the docs

• DatetimeIndex is now in the API documentation, see the docs

• json_normalize() is a new method to allow you to create a flat table from semi-structured JSON data. See the docs (GH1067)

• Added PySide support for the qtpandas DataFrameModel and DataFrameWidget.

• Python csv parser now supports usecols (GH4335)

• Frequencies gained several new offsets:
  — LastWeekOfMonth (GH4637)
  — FY5253, and FY5253Quarter (GH4511)

• DataFrame has a new interpolate method, similar to Series (GH4434, GH1892)

```python
In [93]: df = pd.DataFrame({
'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
       'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})

Out[94]:
```

Additionally, the method argument to interpolate has been expanded to include 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'piecewise_polynomial', 'pchip', 'polynomial', 'spline' The new methods require scipy. Consult the Scipy reference guide and documentation for more information about when the various methods are appropriate. See the docs.

Interpolate now also accepts a limit keyword argument. This works similar to fillna's limit:
In [95]: ser = pd.Series([1, 3, np.nan, np.nan, np.nan, 11])

In [96]: ser.interpolate(limit=2)

Out[96]:
0  1.0  
1  3.0  
2  5.0  
3  7.0  
4  NaN  
5  11.0  
dtype: float64

• Added wide_to_long panel data convenience function. See the docs.

In [97]: np.random.seed(123)

In [98]: df = pd.DataFrame({
    ...:   "A1970" : {0 : "a", 1 : "b", 2 : "c"},
    ...:
    "A1980" : {0 : "d", 1 : "e", 2 : "f"},
    "B1970" : {0 : 2.5, 1 : 1.2, 2 : 7},
    "B1980" : {0 : 3.2, 1 : 1.3, 2 : 1},
    "X" : dict(zip(range(3), np.random.randn(3)))
    })

In [99]: df["id"] = df.index

In [100]: df

Out[100]:
0     a     d  2.5000  3.2000 -1.085631 0
1     b     e  1.1991  1.2997  0.997345 1
2     c     f  0.6997  0.1997  0.282978 2

In [101]: pd.wide_to_long(df, ["A", "B"], i="id", j="year")

Out[101]:
   id year  X  A  B
0  1970 -1.0856  a  2.5
1  1970  0.997345  b  1.2
2  1970  0.282978  c  0.7
0  1980 -1.085631  d  3.2
1  1980  0.997345  e  1.3
2  1980  0.282978  f  0.1

• to_csv now takes a date_format keyword argument that specifies how output datetime objects should be formatted. Datetimes encountered in the index, columns, and values will all have this formatting applied. (GH4313)
• DataFrame.plot will scatter plot x versus y by passing kind='scatter' (GH2215)
• Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. (GH5271)
Experimental

- The new `eval()` function implements expression evaluation using `numexpr` behind the scenes. This results in large speedups for complicated expressions involving large DataFrames/Series. For example,

```python
In [102]: nrows, ncols = 20000, 100

In [103]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols))
       .....:     for _ in range(4)]
       .....:

# eval with NumExpr backend
In [104]: %timeit pd.eval('df1 + df2 + df3 + df4')
5.35 ms +- 188 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

# pure Python evaluation
In [105]: %timeit df1 + df2 + df3 + df4
12.5 ms +- 1.13 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

For more details, see the [docs](#).

- Similar to pandas.eval, DataFrame has a new DataFrame.eval method that evaluates an expression in the context of the DataFrame. For example,

```python
In [106]: df = pd.DataFrame(np.random.randn(10, 2), columns=['a', 'b'])

In [107]: df.eval('a + b')
```

```
Out[107]:
     a   b
0  0.1  0.2
1  0.3  0.4
2  0.5  0.6
3  0.7  0.8
4  0.9  1.0
```
```
dtype: float64
```

- `query()` method has been added that allows you to select elements of a DataFrame using a natural query syntax nearly identical to Python syntax. For example,

```python
In [108]: n = 20

In [109]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=['a', 'b', 'c'])

In [110]: df.query('a < b < c')
```

```
Out[110]:
     a   b   c
10  11  15  16
11  15  16  19
```

selects all the rows of df where `a < b < c` evaluates to True. For more details see the [docs](#).
- `pd.read_msgpack()` and `pd.to_msgpack()` are now a supported method of serialization of arbitrary pandas (and python objects) in a lightweight portable binary format. See the docs

**Warning:** Since this is an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

```python
df = pd.DataFrame(np.random.rand(5, 2), columns=['A', 'B'])
df.to_msgpack('foo.msg')
pd.read_msgpack('foo.msg')

s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))
pd.to_msgpack('foo.msg', df, s)
pd.read_msgpack('foo.msg')
```

You can pass `iterator=True` to iterator over the unpacked results

```python
for o in pd.read_msgpack('foo.msg', iterator=True):
    print(o)
```

- `pandas.io.gbq` provides a simple way to extract from, and load data into, Google's BigQuery Data Sets by way of pandas DataFrames. BigQuery is a high performance SQL-like database service, useful for performing ad-hoc queries against extremely large datasets. See the docs

```python
from pandas.io import gbq

# A query to select the average monthly temperatures in the
# in the year 2000 across the USA. The dataset,
# publicata:samples.gsod, is available on all BigQuery accounts,
# and is based on NOAA gsod data.

query = """SELECT station_number as STATION,
month as MONTH, AVG(mean_temp) as MEAN_TEMP
FROM publicdata:samples.gsod
WHERE YEAR = 2000
GROUP BY STATION, MONTH
ORDER BY STATION, MONTH ASC"

# Fetch the result set for this query

# Your Google BigQuery Project ID
# To find this, see your dashboard:
# https://console.developers.google.com/iam-admin/projects?authuser=0
projectid = 'xxxxxxxxx'
df = gbq.read_gbq(query, project_id=projectid)

# Use pandas to process and reshape the dataset

df2 = df.pivot(index='STATION', columns='MONTH', values='MEAN_TEMP')
df3 = pd.concat([df2.min(), df2.mean(), df2.max()],
                    axis=1, keys=['Min Temp', 'Mean Temp', 'Max Temp'])
```

The resulting DataFrame is:
Warning: To use this module, you will need a BigQuery account. See <https://cloud.google.com/products/big-query> for details.

As of 10/10/13, there is a bug in Google’s API preventing result sets from being larger than 100,000 rows. A patch is scheduled for the week of 10/14/13.

### Internal refactoring

In 0.13.0 there is a major refactor primarily to subclass \texttt{Series} from \texttt{NDFrame}, which is the base class currently for \texttt{DataFrame} and \texttt{Panel}, to unify methods and behaviors. Series formerly subclassed directly from \texttt{ndarray}. (GH4080, GH3862, GH816)

Warning: There are two potential incompatibilities from < 0.13.0

- Using certain numpy functions would previously return a \texttt{Series} if passed a \texttt{Series} as an argument. This seems only to affect \texttt{np.ones_like}, \texttt{np.empty_like}, \texttt{np.diff} and \texttt{np.where}. These now return \texttt{ndarrays}.

<table>
<thead>
<tr>
<th>In [111]:</th>
<th>\texttt{s} = pd.Series([1, 2, 3, 4])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy Usage</td>
<td></td>
</tr>
<tr>
<td>In [112]:</td>
<td>\texttt{np.ones_like(s)}</td>
</tr>
<tr>
<td>Out[112]:</td>
<td>array([1, 1, 1, 1])</td>
</tr>
<tr>
<td>In [113]:</td>
<td>\texttt{np.diff(s)}</td>
</tr>
<tr>
<td>Out[113]:</td>
<td>array([1, 1, 1])</td>
</tr>
<tr>
<td>In [114]:</td>
<td>\texttt{np.where(s &gt; 1, s, np.nan)}</td>
</tr>
<tr>
<td>Out[114]:</td>
<td>array([nan, 2., 3., 4.])</td>
</tr>
</tbody>
</table>

| Pandas Usage |
| In [115]: | pd.Series(1, index=s.index) |
| Out[115]: | 0 1 |
| 1 1 |
2 1
dtype: int64

In [116]: s.diff()
Out[116]:
0  NaN
1  1.0
2  1.0
3  1.0
dtype: float64

In [117]: s.where(s > 1)
Out[117]:
0  NaN
1  2.0
2  3.0
3  4.0
dtype: float64

• Passing a Series directly to a cython function expecting an ndarray type will no long work directly, you must pass Series.values, See Enhancing Performance
• Series(0.5) would previously return the scalar 0.5, instead this will return a 1-element Series
• This change breaks rpy2<=2.3.8. an Issue has been opened against rpy2 and a workaround is detailed in GH5698. Thanks @JanSchulz.

• Pickle compatibility is preserved for pickles created prior to 0.13. These must be unpickled with pd. read_pickle, see Pickling.
• Refactor of series.py/frame.py/panel.py to move common code to generic.py
  – added _setup_axes to created generic NDFrame structures
  – moved methods
    * from_axes,_wrap_array,axes,ix,loc,iloc,shape,empty,swapaxes,transpose,drop
    * __iter__,keys,__contains__,__len__,__neg__,__invert__
    * convert_objects,as_blocks,as_matrix,values
    * __getstate__,__setstate__ (compat remains in frame/panel)
    * __getattr__,__setattr__
    * _indexed_same, reindex_like,align,where,mask
    * fillna,replace (Series replace is now consistent with DataFrame)
    * filter (also added axis argument to selectively filter on a different axis)
    * reindex,reindex_axis,take
    * truncate (moved to become part of NDFrame)
• These are API changes which make Panel more consistent with DataFrame
  – swapaxes on a Panel with the same axes specified now return a copy
  – support attribute access for setting
filter supports the same API as the original DataFrame filter
• Reindex called with no arguments will now return a copy of the input object
• TimeSeries is now an alias for Series. the property is.time_series can be used to distinguish (if desired)
• Refactor of Sparse objects to use BlockManager
  – Created a new block type in internals, SparseBlock, which can hold multi-dtypes and is non-consolidatable. SparseSeries and SparseDataFrame now inherit more methods from there hierarchy (Series/DataFrame), and no longer inherit from SparseArray (which instead is the object of the SparseBlock)
  – Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
  – Operations on sparse structures within DataFrames should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
  – enable setitem on SparseSeries for boolean/integer/slices
  – SparsePanels implementation is unchanged (e.g. not using BlockManager, needs work)
• added ftypes method to Series/DataFrame, similar to dtypes, but indicates if the underlying is sparse/dense (as well as the dtype)

• All NDFrame objects can now use __finalize__() to specify various values to propagate to new objects from an existing one (e.g. name in Series will follow more automatically now)
• Internal type checking is now done via a suite of generated classes, allowing isinstance(value, klass) without having to directly import the klass, courtesy of @jtratner
• Bug in Series update where the parent frame is not updating its cache based on changes (GH4080) or types (GH3217), fillna (GH3386)
• Indexing with dtype conversions fixed (GH4463, GH4204)
• Refactor Series.reindex to core/generic.py (GH4604, GH4618), allow method= in reindexing on a Series to work
• Series.copy no longer accepts the order parameter and is now consistent with NDFrame copy
• Refactor rename methods to core/generic.py; fixes Series.rename for (GH4605), and adds rename with the same signature for Panel
• Refactor clip methods to core/generic.py (GH4798)
• Refactor of _get_numeric_data/_get_bool_data to core/generic.py, allowing Series/Panel functionality
• Series (for index) / Panel (for items) now allow attribute access to its elements (GH1903)

```python
In [118]: s = pd.Series([1, 2, 3], index=list('abc'))
In [119]: s.b
Out[119]: 2
In [120]: s.a = 5
In [121]: s
Out[121]:
a    5
b    2
```
(continues on next page)
Bug fixes

- HDFStore
  - raising an invalid TypeError rather than ValueError when appending with a different block ordering (GH4096)
  - read_hdf was not respecting as passed mode (GH4504)
  - appending a 0-len table will work correctly (GH4273)
  - to_hdf was raising when passing both arguments append and table (GH4584)
  - reading from a store with duplicate columns across dtypes would raise (GH4767)
  - Fixed a bug where ValueError wasn’t correctly raised when column names weren’t strings (GH4956)
  - A zero length series written in Fixed format not deserializing properly. (GH4708)
  - Fixed decoding perf issue on pyt3 (GH5441)
  - Validate levels in a MultiIndex before storing (GH5527)
  - Correctly handle data_columns with a Panel (GH5717)
- Fixed bug in tslib.tz_convert(vals, tz1, tz2): it could raise IndexError exception while trying to access trans[pos + 1] (GH4496)
- The by argument now works correctly with the layout argument (GH4102, GH4014) in *.hist plotting methods
- Fixed bug in PeriodIndex.map where using str would return the str representation of the index (GH4136)
- Fixed test failure test_time_series_plot_color_with_empty_kwargs when using custom matplotlib default colors (GH4345)
- Fix running of stata IO tests. Now uses temporary files to write (GH4353)
- Fixed an issue where DataFrame.sum was slower than DataFrame.mean for integer valued frames (GH4365)
- read_html tests now work with Python 2.6 (GH4351)
- Fixed bug where network testing was throwing NameError because a local variable was undefined (GH4381)
- In to_json, raise if a passed orient would cause loss of data because of a duplicate index (GH4359)
- In to_json, fix date handling so milliseconds are the default timestamp as the docstring says (GH4362).
- as_index is no longer ignored when doing groupby apply (GH4648, GH3417)
- JSON NaT handling fixed, NaTs are now serialized to null (GH4498)
- Fixed JSON handling of escapable characters in JSON object keys (GH4593)
- Fixed passing keep_default_na=False when na_values=None (GH4318)
- Fixed bug with values raising an error on a DataFrame with duplicate columns and mixed dtypes, surfaced in (GH4377)
- Fixed bug with duplicate columns and type conversion in read_json when orient='split' (GH4377)
• Fixed JSON bug where locales with decimal separators other than ‘.’ threw exceptions when encoding / decoding certain values. (GH4918)

• Fix .iat indexing with a PeriodIndex (GH4390)

• Fixed an issue where PeriodIndex joining with self was returning a new instance rather than the same instance (GH4379); also adds a test for this for the other index types

• Fixed a bug with all the dtypes being converted to object when using the CSV parser with the usecols parameter (GH3192)

• Fix an issue in merging blocks where the resulting DataFrame had partially set _ref_locs (GH4403)

• Fixed an issue where hist subplots were being overwritten when they were called using the top level matplotlib API (GH4408)

• Fixed a bug where calling Series.astype(str) would truncate the string (GH4405, GH4437)

• Fixed a py3 compat issue where bytes were being repr’d as tuples (GH4455)

• Fixed Panel attribute naming conflict if item is named ‘a’ (GH3440)

• Fixed an issue where duplicate indexes were raising when plotting (GH4486)

• Fixed an issue where cumsum and cumprod didn’t work with bool dtypes (GH4170, GH4440)

• Fixed Panel slicing issued in xs that was returning an incorrect dimmed object (GH4016)

• Fix resampling bug where custom reduce function not used if only one group (GH3849, GH4494)

• Fixed Panel assignment with a transposed frame (GH3830)

• Raise on set indexing with a Panel and a Panel as a value which needs alignment (GH3777)

• frozenset objects now raise in the Series constructor (GH4482, GH4480)

• Fixed issue with sorting a duplicate MultiIndex that has multiple dtypes (GH4516)

• Fixed bug in DataFrame.set_values which was causing name attributes to be lost when expanding the index. (GH3742, GH4039)

• Fixed issue where individual names, levels and labels could be set on MultiIndex without validation (GH3714, GH4039)

• Fixed (GH3334) in pivot_table. Margins did not compute if values is the index.

• Fix bug in having a rhs of np.timedelta64 or np.offsets.DateOffset when operating with datetimes (GH4532)

• Fix arithmetic with series/datetimeindex and np.timedelta64 not working the same (GH4134) and buggy timedelta in NumPy 1.6 (GH4135)

• Fix bug in pd.read_clipboard on windows with PY3 (GH4561); not decoding properly

• tslib.get_period_field() and tslib.get_period_field_arr() now raise if code argument out of range (GH4519, GH4520)

• Fix boolean indexing on an empty series loses index names (GH4235), infer_dtype works with empty arrays.

• Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue (GH3317)

• Fixed issue where DataFrame.apply was reraising exceptions incorrectly (causing the original stack trace to be truncated).

• Fix selection with ix/loc and non_unique selectors (GH4619)
• Fix assignment with iloc/loc involving a dtype change in an existing column (GH4312, GH5702) have internal setitem_with_indexer in core/indexing to use Block.setitem

• Fixed bug where thousands operator was not handled correctly for floating point numbers in csv_import (GH4322)

• Fix an issue with CacheableOffset not properly being used by many DateOffset; this prevented the DateOffset from being cached (GH4609)

• Fix boolean comparison with a DataFrame on the lhs, and a list/tuple on the rhs (GH4576)

• Fix error/dtype conversion with setitem of None on Series/DataFrame (GH4667)

• Fix decoding based on a passed in non-default encoding in pd.read_stata (GH4626)

• Fix DataFrame.from_records with a plain-vanilla ndarray. (GH4727)

• Fix some inconsistencies with Index.rename and MultiIndex.rename, etc. (GH4718, GH4628)

• Bug in using iloc/loc with a cross-sectional and duplicate indices (GH4726)

• Bug with using QUOTE_NONE with to_csv causing Exception. (GH4328)

• Bug with Series indexing not raising an error when the right-hand-side has an incorrect length (GH2702)

• Bug in MultiIndexing with a partial string selection as one part of a MultiIndex (GH4758)

• Bug with reindexing on the index with a non-unique index will now raise ValueError (GH4746)

• Bug in setting with loc/ix a single indexer with a MultiIndex axis and a NumPy array, related to (GH3777)

• Bug in concatenation with duplicate columns across dtypes not merging with axis=0 (GH4771, GH4975)

• Bug in iloc with a slice index failing (GH4771)

• Incorrect error message with no colspecs or width in read_fwf. (GH4774)

• Fix bugs in indexing in a Series with a duplicate index (GH4548, GH4550)

• Fixed bug with reading compressed files with read_fwf in Python 3. (GH3963)

• Fixed an issue with a duplicate index and assignment with a dtype change (GH4686)


• Fixed an issue related to ticklocs/ticklabels with log scale bar plots across different versions of matplotlib (GH4789)

• Suppressed DeprecationWarning associated with internal calls issued by repr() (GH4391)

• Fixed an issue with a duplicate index and duplicate selector with .loc (GH4825)

• Fixed an issue with DataFrame.sort_index where, when sorting by a single column and passing a list for ascending, the argument for ascending was being interpreted as True (GH4839, GH4846)

• Fixed Panel.tshift not working. Added freq support to Panel.shift (GH4853)

• Fix an issue in TextFileReader w/ Python engine (i.e. PythonParser) with thousands != “,” (GH4596)

• Bug in getitem with a duplicate index when using where (GH4879)

• Fix Type inference code coerces float column into datetime (GH4601)

• Fixed _ensure_numeric does not check for complex numbers (GH4902)

• Fixed a bug in Series.hist where two figures were being created when the by argument was passed (GH4112, GH4113).
• Fixed a bug in convert_objects for > 2 ndims (GH4937)
• Fixed a bug in DataFrame/Panel cache insertion and subsequent indexing (GH4939, GH5424)
• Fixed string methods for FrozenNDArray and FrozenList (GH4929)
• Fixed a bug with setting invalid or out-of-range values in indexing enlargement scenarios (GH4940)
• Tests for fillna on empty Series (GH4346), thanks @immerrrr
• Fixed copy() to shallow copy axes/indices as well and thereby keep separate metadata. (GH4202, GH4830)
• Fixed skiprows option in Python parser for read_csv (GH4382)
• Fixed bug preventing cut from working with np.inf levels without explicitly passing labels (GH3415)
• Fixed wrong check for overlapping in DatetimeIndex.union (GH4564)
• Fixed conflict between thousands separator and date parser in csv_parser (GH4678)
• Fix appending when dtypes are not the same (error showing mixing float/np.datetime64) (GH4993)
• Fix repr for DateOffset. No longer show duplicate entries in kwds. Removed unused offset fields. (GH4638)
• Fixed wrong index name during read_csv if using usecols. Applies to c parser only. (GH4201)
• Timestamp objects can now appear in the left hand side of a comparison operation with a Series or DataFrame object (GH4982).
• Fix a bug when indexing with np.nan via iloc/loc (GH5016)
• Fixed a bug where low memory c parser could create different types in different chunks of the same file. Now coerces to numerical type or raises warning. (GH3866)
• Fix a bug where reshaping a Series to its own shape raised TypeError (GH4554) and other reshaping issues.
• Bug in setting with ix/loc and a mixed int/string index (GH4544)
• Make sure series-series boolean comparisons are label based (GH4947)
• Bug in multi-level indexing with a Timestamp partial indexer (GH4294)
• Tests/fix for MultiIndex construction of an all-nan frame (GH4078)
• Fixed a bug where read_html() wasn’t correctly inferring values of tables with commas (GH5029)
• Fixed a bug where read_html() wasn’t providing a stable ordering of returned tables (GH4770, GH5029).
• Fixed a bug where read_html() was incorrectly parsing when passed index_col=0 (GH5066).
• Fixed a bug where read_html() was incorrectly inferring the type of headers (GH5048).
• Fixed a bug where DatetimeIndex joins with PeriodIndex caused a stack overflow (GH3899).
• Fixed a bug where groupby objects didn’t allow plots (GH5102).
• Fixed a bug where groupby objects weren’t tab-completing column names (GH5102).
• Fixed a bug where groupby.plot() and friends were duplicating figures multiple times (GH5102).
• Provide automatic conversion of object dtypes on fillna, related (GH5103)
• Fixed a bug where default options were being overwritten in the option parser cleaning (GH5121).
• Treat a list/ndarray identically for iloc indexing with list-like (GH5006)
• Fix MultiIndex.get_level_values() with missing values (GH5074)
• Fix bound checking for Timestamp() with datetime64 input (GH4065)
• Fix a bug where TestReadHtml wasn’t calling the correct read_html() function (GH5150).
• Fix a bug with NDFrame.replace() which made replacement appear as though it was (incorrectly) using regular expressions (GH5143).
• Fix better error message for to_datetime (GH4928)
• Made sure different locales are tested on travis-ci (GH4918). Also adds a couple of utilities for getting locales and setting locales with a context manager.
• Fixed segfault on isnull(MultiIndex) (now raises an error instead) (GH5123, GH5125)
• Allow duplicate indices when performing operations that align (GH5185, GH5639)
• Compound dtypes in a constructor raise NotImplementedError (GH5191)
• Bug in comparing duplicate frames (GH4421) related
• Bug in describe on duplicate frames
• Bug in to_datetime with a format and coerce=True not raising (GH5195)
• Bug in loc setting with multiple indexers and a rhs of a Series that needs broadcasting (GH5206)
• Fixed bug where inplace setting of levels or labels on MultiIndex would not clear cached values property and therefore return wrong values. (GH5215)
• Fixed bug where filtering a grouped DataFrame or Series did not maintain the original ordering (GH4621).
• Fixed Period with a business date freq to always roll-forward if on a non-business date. (GH5203)
• Fixed bug in Excel writers where frames with duplicate column names weren’t written correctly. (GH5235)
• Fixed issue with drop and a non-unique index on Series (GH5248)
• Fixed segfault in C parser caused by passing more names than columns in the file. (GH5156)
• Fix Series.isin with date/time-like dtypes (GH5021)
• C and Python Parser can now handle the more common MultiIndex column format which doesn’t have a row for index names (GH4702)
• Bug when trying to use an out-of-bounds date as an object dtype (GH5312)
• Bug when trying to display an embedded PandasObject (GH5324)
• Allows operating of Timestamps to return a datetime if the result is out-of-bounds related (GH5312)
• Fix return value/type signature of initObjToJSON() to be compatible with numpy’s import_array() (GH5334, GH5326)
• Bug when renaming then set_index on a DataFrame (GH5344)
• Test suite no longer leaves around temporary files when testing graphics. (GH5347) (thanks for catching this @yarikoptic!)
• Fixed html tests on win32. (GH4580)
• Make sure that head/tail are iloc based, (GH5370)
• Fixed bug for PeriodIndex string representation if there are 1 or 2 elements. (GH5372)
• The GroupBy methods transform and filter can be used on Series and DataFrames that have repeated (non-unique) indices. (GH4620)
• Fix empty series not printing name in repr (GH4651)
• Make tests create temp files in temp directory by default. (GH5419)
• pd.to_timedelta of a scalar returns a scalar (GH5410)
• `pd.to_timedelta` accepts `NaN` and `NaT`, returning `NaT` instead of raising (GH5437)
• performance improvements in `isnull` on larger size pandas objects
• Fixed various `setitem` with 1d ndarray that does not have a matching length to the indexer (GH5508)
• Bug in `getitem` with a MultiIndex and `iloc` (GH5528)
• Bug in `delitem` on a Series (GH5542)
• Bug fix in apply when using custom function and objects are not mutated (GH5545)
• Bug in selecting from a non-unique index with `loc` (GH5553)
• Bug in `groupby` returning non-consistent types when user function returns a `None`, (GH5592)
• Work around regression in `numpy 1.7.0` which erroneously raises `IndexError` from `ndarray.item` (GH5666)
• Bug in repeated indexing of object with resultant non-unique index (GH5678)
• Bug in `fillna` with Series and a passed series/dict (GH5703)
• Bug in `groupby` transform with a datetime-like grouper (GH5712)
• Bug in `MultiIndex` selection in PY3 when using certain keys (GH5725)
• Row-wise concat of differing dtypes failing in certain cases (GH5754)

Contributors

A total of 77 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Agustín Herranz +
• Alex Gaudio +
• Alex Rothberg +
• Andreas Klostermann +
• Andreas Wurl +
• Andy Hayden
• Ben Alex +
• Benedikt Sauer +
• Brad Buran
• Caleb Epstein +
• Chang She
• Christopher Whelan
• DSM +
• Dale Jung +
• Dan Birken
• David Rasch +
• Dieter Vandenbussche
• Gabi Davar +
• Garrett Drapala
• Goyo +
• Greg Reda +
• Ivan Smirnov +
• Jack Kelly +
• Jacob Schaer +
• Jan Schulz +
• Jeff Tratner
• Jeffrey Tratner
• John McNamara +
• John W. O’Brien +
• Joris Van den Bossche
• Justin Bozonier +
• Kelsey Jordahl
• Kevin Stone
• Kieran O’Mahony
• Kyle Hausmann +
• Kyle Kelley +
• Kyle Meyer
• Mike Kelly
• Mortada Mehyar +
• Nick Foti +
• Olivier Harris +
• Ondřej Čertík +
• PKEuS
• Phillip Cloud
• Pierre Haessig +
• Richard T. Guy +
• Roman Pekar +
• Roy Hyunjin Han
• Skipper Seabold
• Sten +
• Thomas A Caswell +
• Thomas Kluyver
• Tiago Requeijo +
• TomAugspurger
5.19 Version 0.12

5.19.1 Version 0.12.0 (July 24, 2013)

This is a major release from 0.11.0 and includes several new features and enhancements along with a large number of bug fixes.

Highlights include a consistent I/O API naming scheme, routines to read html, write MultiIndexes to csv files, read & write STATA data files, read & write JSON format files, Python 3 support for HDFStore, filtering of groupby expressions via filter, and a revamped replace routine that accepts regular expressions.
API changes

• The I/O API is now much more consistent with a set of top level reader functions accessed like `pd.read_csv()` that generally return a pandas object.
  
  - `read_csv`
  - `read_excel`
  - `read_hdf`
  - `read_sql`
  - `read_json`
  - `read_html`
  - `read_stata`
  - `read_clipboard`

  The corresponding writer functions are object methods that are accessed like `df.to_csv()`
  
  - `to_csv`
  - `to_excel`
  - `to_hdf`
  - `to_sql`
  - `to_json`
  - `to_html`
  - `to_stata`
  - `to_clipboard`

• Fix modulo and integer division on Series,DataFrames to act similarly to float dtypes to return `np.nan` or `np.inf` as appropriate (GH3590). This correct a numpy bug that treats integer and float dtypes differently.

```
In [1]: p = pd.DataFrame({"first": [4, 5, 8], "second": [0, 0, 3]})
In [2]: p % 0
Out[2]:
   first  second
   0    NaN    NaN
   1    NaN    NaN
   2    NaN    NaN
In [3]: p % p
Out[3]:
   first  second
   0  0.0    NaN
   1  0.0    NaN
   2  0.0    0.0
In [4]: p / p
Out[4]:
   first  second
   0    1.0    NaN
```

(continues on next page)
Add squeeze keyword to groupby to allow reduction from DataFrame -> Series if groups are unique. This is a Regression from 0.10.1. We are reverting back to the prior behavior. This means groupby will return the same shaped objects whether the groups are unique or not. Revert this issue (GH2893) with (GH3596).

```
In[2]: df2 = pd.DataFrame([["val1": 1, "val2": 20],
                         ["val2": 1, "val2": 19],
                         ["val1": 1, "val2": 27],
                         ["val1": 1, "val2": 12]])
```

```
In[3]: def func(dataf):
    ...: return dataf["val2"] - dataf["val2"].mean()
    ...
```

```
In[4]: # squeezing the result frame to a series (because we have unique groups)
    ...: df2.groupby("val1", squeeze=True).apply(func)
```

```
Out[4]:
   0   1   2
0  0.5 7.5
1 -0.5 7.5
2  7.5 7.5
3 -7.5 7.5
Name: 1, dtype: float64
```

```
In[5]: # no squeezing (the default, and behavior in 0.10.1)
    ...: df2.groupby("val1").apply(func)
```

```
Out[5]:
     val2
val1   0  1  2
  1  0.5 7.5 7.5
```

Raise on iloc when boolean indexing with a label based indexer mask e.g. a boolean Series, even with integer labels, will raise. Since iloc is purely positional based, the labels on the Series are not alignable (GH3631)

This case is rarely used, and there are plenty of alternatives. This preserves the iloc API to be purely positional based.

```
In[6]: df = pd.DataFrame(range(5), index=list("ABCDE"), columns=["a"])
```

```
In[7]: mask = df.a % 2 == 0
```

```
In[8]: mask
```

```
Out[8]:
   A  True
   B False
```
C  True
D  False
E  True
Name: a, dtype: bool

# this is what you should use
In [9]: df.loc[mask]
Out[9]:
   a
A  0
C  2
E  4

# this will work as well
In [10]: df.iloc[mask.values]
Out[10]:
   a
A  0
C  2
E  4

df.iloc[mask] will raise a ValueError

• The raise_on_error argument to plotting functions is removed. Instead, plotting functions raise a TypeError when the dtype of the object is object to remind you to avoid object arrays whenever possible and thus you should cast to an appropriate numeric dtype if you need to plot something.

• Add colormap keyword to DataFrame plotting methods. Accepts either a matplotlib colormap object (ie, matplotlib.cm.jet) or a string name of such an object (ie, ‘jet’). The colormap is sampled to select the color for each column. Please see Colormaps for more information. (GH3860)

• DataFrame.interpolate() is now deprecated. Please use DataFrame.fillna() and DataFrame.replace() instead. (GH3582, GH3675, GH3676)

• the method and axis arguments of DataFrame.replace() are deprecated

• DataFrame.replace ’s infer_types parameter is removed and now performs conversion by default. (GH3907)

• Add the keyword allow_duplicates to DataFrame.insert to allow a duplicate column to be inserted if True, default is False (same as prior to 0.12) (GH3679)

• Implement __nonzero__ for NDFrame objects (GH3691, GH3696)

• IO api
  – added top-level function read_excel to replace the following. The original API is deprecated and will be removed in a future version

    from pandas.io.parsers import ExcelFile

    xls = ExcelFile("path_to_file.xls")
    xls.parse("Sheet1", index_col=None, na_values=["NA"])

    With
**import pandas as pd**

```python
pd.read_excel("path_to_file.xls", "Sheet1", index_col=None, na_values=["NA"])
```

- added top-level function `read_sql` that is equivalent to the following

```python
from pandas.io.sql import read_frame
read_frame(...)
```

- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Do not allow asctypes on `datetime64[ns]` except to `object`, and `timedelta64[ns]` to `object/int` (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when performed on a `Series` and return an empty `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of slice objects:
  - sum, prod, mean, std, var, skew, kurt, corr, and cov
- `read_html` now defaults to `None` when reading, and falls back on `bs4 + html5lib` when `lxml` fails to parse. a list of parsers to try until success is also valid
- The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the base class for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)
  - New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)

### IO enhancements

- `pd.read_html()` can now parse HTML strings, files or urls and return DataFrames, courtesy of @cpcloud. (GH3477, GH3605, GH3606, GH3616). It works with a single parser backend: BeautifulSoup4 + html5lib. See the docs

You can use `pd.read_html()` to read the output from `DataFrame.to_html()` like so

```python
In [11]: df = pd.DataFrame({"a": range(3), "b": list("abc")})

In [12]: print(df)
   a  b
0  0  a
1  1  b
2  2  c

In [13]: html = df.to_html()

In [14]: alist = pd.read_html(html, index_col=0)

In [15]: print(df == alist[0])
   a  b
0  True True
```

(continues on next page)
Note that `alist` here is a Python list so `pd.read_html()` and `DataFrame.to_html()` are not inverses.

- `pd.read_html()` no longer performs hard conversion of date strings (GH3656).

**Warning:** You may have to install an older version of BeautifulSoup4, *See the installation docs*

- Added module for reading and writing Stata files: `pandas.io.stata` (GH1512) accessible via `read_stata` top-level function for reading, and `to_stata` DataFrame method for writing, *See the docs*
- Added module for reading and writing json format files: `pandas.io.json` accessible via `read_json` top-level function for reading, and `to_json` DataFrame method for writing, *See the docs* various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
- MultiIndex column support for reading and writing csv format files
  - The `header` option in `read_csv` now accepts a list of the rows from which to read the index.
  - The option, `tupleize_cols` can now be specified in both `to_csv` and `read_csv` to provide compatibility for the pre 0.12 behavior of writing and reading MultiIndex columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a MultiIndex column.
  
  Note: The default behavior in 0.12 remains unchanged from prior versions, but starting with 0.13, the default to write and read MultiIndex columns will be in the new format. (GH3571, GH1651, GH3141)
  - If an `index_col` is not specified (e.g. you don’t have an index, or wrote it with `df.to_csv(..., index=False)`), then any names on the columns index will be *lost*.

```python
In [16]: from pandas._testing import makeCustomDataFrame as mkdf

In [17]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [18]: df.to_csv("mi.csv")

In [19]: print(open("mi.csv").read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2

In [20]: pd.read_csv("mi.csv", header=[0, 1, 2, 3], index_col=[0, 1])
```

```python
Out[20]:
C0 C_l0_g0 C_l0_g1 C_l0_g2
C1 C_l1_g0 C_l1_g1 C_l1_g2
C2 C_l2_g0 C_l2_g1 C_l2_g2
```
• Support for HDFStore (via PyTables 3.0.0) on Python3

• Iterator support via `read_hdf` that automatically opens and closes the store when iteration is finished. This is only for `tables`

```python
In [25]: path = 'store_iterator.h5'

In [26]: pd.DataFrame(np.random.randn(10, 2)).to_hdf(path, 'df', table=True)

In [27]: for df in pd.read_hdf(path, 'df', chunksize=3):
   ....:     print(df)
   ....:
    0  1
    0  0.713216 -0.778461
    1 -0.661062  0.862877
    2  0.344342  0.149565
    0  1
    3 -0.626968 -0.875772
    4 -0.930687 -0.218983
    5  0.949965 -0.442354
    0  1
    6 -0.402985  1.111358
    7 -0.241527 -0.670477
    8  0.049355  0.632633
    0  1
    9 -1.502767 -1.225492
```

• `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters

Other enhancements

• `DataFrame.replace()` now allows regular expressions on contained Series with object dtype. See the examples section in the regular docs Replacing via String Expression

For example you can do

```python
In [21]: df = pd.DataFrame({'a': list('ab..'), 'b': [1, 2, 3, 4]})

In [22]: df.replace(regex=r'^\s*\.' , value=np.nan)
Out[22]:
    a  b
0  a  1
1  b  2
```

(continues on next page)
to replace all occurrences of the string '. ' with zero or more instances of surrounding white space with NaN.

Regular string replacement still works as expected. For example, you can do

```python
In [23]: df.replace(". ", np.nan)
```

```plaintext
Out[23]:
a b
0 a 1
1 b 2
2 NaN 3
3 NaN 4
```

to replace all occurrences of the string '. ' with NaN.

- `pd.melt()` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame.
- `pd.set_option()` now allows N option, value pairs (GH3667).

Let’s say that we had an option 'a.b' and another option 'b.c'. We can set them at the same time:

```python
In [31]: pd.get_option('a.b')
```

```plaintext
Out[31]: 2
```

```python
In [32]: pd.get_option('b.c')
```

```plaintext
Out[32]: 3
```

```python
In [33]: pd.set_option('a.b', 1, 'b.c', 4)
```

```python
In [34]: pd.get_option('a.b')
```

```plaintext
Out[34]: 1
```

```python
In [35]: pd.get_option('b.c')
```

```plaintext
Out[35]: 4
```

- The `filter` method for group objects returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```python
In [24]: sf = pd.Series([1, 1, 2, 3, 3, 3])
```

```python
In [25]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
```

```plaintext
Out[25]:
3  3
4  3
5  3
dtype: int64
```

The argument of `filter` must a function that, applied to the group as a whole, returns `True` or `False`. Another useful operation is filtering out elements that belong to groups with only a couple members.
In [26]: dff = pd.DataFrame({"A": np.arange(8), "B": list("aabbbbcc")})
In [27]: dff.groupby("B").filter(lambda x: len(x) > 2)
Out[27]:
   A  B
2  2  b
3  3  b
4  4  b
5  5  b

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

In [28]: dff.groupby("B").filter(lambda x: len(x) > 2, dropna=False)
Out[28]:
   A  B
0  NaN NaN
1  NaN NaN
2  2.0  b
3  3.0  b
4  4.0  b
5  5.0  b
6  NaN NaN
7  NaN NaN

- Series and DataFrame hist methods now take a figsize argument (GH3834)
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations (GH3877)
- Timestamp.min and Timestamp.max now represent valid Timestamp instances instead of the default datetime.min and datetime.max (respectively), thanks @SleepingPills
- read_html now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

**Experimental features**

- Added experimental CustomBusinessDay class to support DateOffsets with custom holiday calendars and custom weekmasks. (GH2301)

Note: This uses the numpy.busdaycalendar API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.
In [32]: holidays = ["2012-05-01", datetime(2013, 5, 1), np.datetime64("2014-05-01")]

In [33]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [34]: dt = datetime(2013, 4, 30)

In [35]: print(dt + 2 * bday_egypt)
   2013-05-05 00:00:00

In [36]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [37]: print(pd.Series(dts.weekday, dts).map(pd.Series("Mon Tue Wed Thu Fri Sat Sun".split())))
   2013-04-30 Tue
   2013-05-02 Thu
   2013-05-05 Sun
   2013-05-06 Mon
   2013-05-07 Tue
Freq: C, dtype: object

Bug fixes

- Plotting functions now raise a TypeError before trying to plot anything if the associated objects have a dtype of object (GH1818, GH3572, GH3911, GH3912), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which eliminates any spurious plots from showing up.

-fillna methods now raise a TypeError if the value parameter is a list or tuple.

- Series.str now supports iteration (GH3638). You can iterate over the individual elements of each string in the Series. Each iteration yields a Series with either a single character at each index of the original Series or NaN. For example,

In [38]: strs = "go", "bow", "joe", "slow"

In [39]: ds = pd.Series(strs)

In [40]: for s in ds.str:
   ....:     print(s)
   ....:
0     g
1     b
2     j
3     s
dtype: object
0     o
1     o
2     o
3     l
dtype: object
0       NaN
The last element yielded by the iterator will be a Series containing the last element of the longest string in the Series with all other elements being NaN. Here since 'slow' is the longest string and there are no other strings with the same length 'w' is the only non-null string in the yielded Series.

- **HDFStore**
  - will retain index attributes (freq,tz,name) on recreation (GH3499)
  - will warn with an AttributeError if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
  - support datelike columns with a timezone as data_columns (GH2852)

- **Non-unique index support clarified** (GH3468).
  - Fix assigning a new index to a duplicate index in a DataFrame would fail (GH3468)
  - Fix construction of a DataFrame with a duplicate index
  - ref_locs support to allow duplicative indices across dtypes, allows iget support to always find the index (even across dtypes) (GH2194)
  - applymap on a DataFrame with a non-unique index now works (removed warning) (GH2786), and fix (GH3230)
  - Fix to_csv to handle non-unique columns (GH3495)
  - Duplicate indexes with getitem will return items in the correct order (GH3455, GH3457) and handle missing elements like unique indices (GH3561)
  - Duplicate indexes with and empty DataFrame.from_records will return a correct frame (GH3562)
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed (GH3602)
  - Allow insert/delete to non-unique columns (GH3679)
  - Non-unique indexing with a slice via loc and friends fixed (GH3659)
  - Allow insert/delete to non-unique columns (GH3679)
- Extend reindex to correctly deal with non-unique indices (GH3679)
- DataFrame.iter_tuples() now works with frames with duplicate column names (GH3873)
- Bug in non-unique indexing via iloc (GH4017); added takeable argument to reindex for location-based taking
- Allow non-unique indexing in series via .ix/.loc and __getitem__ (GH4246)
- Fixed non-unique indexing memory allocation issue with .ix/.loc (GH4280)
- DataFrame.from_records did not accept empty recarrays (GH3682)
- read_html now correctly skips tests (GH3741)
- Fixed a bug where DataFrame.replace with a compiled regular expression in the to_replace argument wasn't working (GH3907)
- Improved network test decorator to catch IOError (and therefore URLError as well). Added with_connectivity_check decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new optional_args decorator factory for decorators. (GH3910, GH3914)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- Series.hist will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)
- Fixed running of tox under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fixed bug in DataFrame.replace where a nested dict wasn’t being iterated over when regex=False (GH4115)
- Fixed bug in the parsing of microseconds when using the format argument in to_datetime (GH4152)
- Fixed bug in PandasAutoDateLocator where invert_xaxis triggered incorrectly MilliSecondLocator (GH3990)
- Fixed bug in plotting that wasn’t raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in DataFrame.plot(kind='kde') (GH4216)
- Fixed bug where Index slices weren’t carrying the name attribute (GH4226)
- Fixed bug in initializing DatetimeIndex with an array of strings in a certain time zone (GH4229)
- Fixed bug where html5lib wasn’t being properly skipped (GH4265)
- Fixed bug where get_data_famafrench wasn’t using the correct file edges (GH4281)

See the full release notes or issue tracker on GitHub for a complete list.
Contributors

A total of 50 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Andy Hayden
- Chang She
- Christopher Whelan
- Damien Garaud
- Dan Allan
- Dan Birken
- Dieter Vandenbussche
- Dražen Lučanin
- Gábor Lipták +
- Jeff Mellen +
- Jeff Tratner +
- Jeffrey Tratner +
- Jonathan deWerd +
- Joris Van den Bossche +
- Juraj Niznan +
- Karmel Allison
- Kelsey Jordahl
- Kevin Stone +
- Kieran O’Mahony
- Kyle Meyer +
- Mike Kelly +
- PKEuS +
- Patrick O’Brien +
- Phillip Cloud
- Richard Hochenberger +
- Skipper Seabold
- SleepingPills +
- Tobias Brandt
- Tom Farnbauer +
- TomAugspurger +
- Trent Hauck +
- Wes McKinney
- Wouter Overmeire
This is a major release from 0.10.1 and includes many new features and enhancements along with a large number of bug fixes. The methods of Selecting Data have had quite a number of additions, and Dtype support is now full-fledged. There are also a number of important API changes that long-time pandas users should pay close attention to.

There is a new section in the documentation, 10 Minutes to Pandas, primarily geared to new users.

There is a new section in the documentation, Cookbook, a collection of useful recipes in pandas (and that we want contributions!).

There are several libraries that are now Recommended Dependencies

Selection choices

Starting in 0.11.0, object selection has had a number of user-requested additions in order to support more explicit location based indexing. pandas now supports three types of multi-axis indexing.

- .loc is strictly label based, will raise KeyError when the items are not found, allowed inputs are:
  - A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index. This use is not an integer position along the index)
  - A list or array of labels ['a', 'b', 'c']
  - A slice object with labels 'a': 'f', (note that contrary to usual python slices, both the start and the stop are included!)
A boolean array

See more at Selection by Label

- .iloc is strictly integer position based (from 0 to length-1 of the axis), will raise IndexError when the requested indices are out of bounds. Allowed inputs are:
  - An integer e.g. 5
  - A list or array of integers [4, 3, 0]
  - A slice object with ints 1:7
  - A boolean array

See more at Selection by Position

- .ix supports mixed integer and label based access. It is primarily label based, but will fallback to integer positional access. .ix is the most general and will support any of the inputs to .loc and .iloc, as well as support for floating point label schemes. .ix is especially useful when dealing with mixed positional and label based hierarchical indexes.

As using integer slices with .ix have different behavior depending on whether the slice is interpreted as position based or label based, it’s usually better to be explicit and use .iloc or .loc.

See more at Advanced Indexing and Advanced Hierarchical.

Selection deprecations

Starting in version 0.11.0, these methods may be deprecated in future versions.

- irow
- icol
- iget_value

See the section Selection by Position for substitutes.

Dtypes

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the dtype keyword, a passed ndarray, or a passed Series), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will NOT be combined. The following example will give you a taste.

```
In [1]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [2]: df1
Out[2]:
   A
0  0.469112
1 -0.282863
2 -1.509058
3 -1.135632
4  1.212112
5 -0.173215
6  0.119209
7 -1.044236
```

(continues on next page)
In [3]: df1.dtypes
Out[3]:
A  float32
dtype: object

In [4]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8), dtype='float16'),
                        'B': pd.Series(np.random.randn(8)),
                        'C': pd.Series(range(8), dtype='uint8')})

In [5]: df2
Out[5]:
   A         B         C
0 -0.861816 -0.424972 0
1  2.105469  0.567020 1
2 -0.494873  0.276232 2
3  1.072266 -1.087401 3
4  0.721680 -0.673690 4
5 -0.706543  0.113648 5
6 -1.040039 -1.478427 6
7  0.271973  0.524988 7

In [6]: df2.dtypes
Out[6]:
A  float16
B  float64
C    uint8
dtype: object

# here you get some upcasting
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [8]: df3
Out[8]:
   A         B         C
0 -0.392704 -0.424972 0.0
1 -2.388332  0.567020 1.0
2 -2.003932  0.276232 2.0
3 -2.003932 -1.087401 3.0
4  1.933792 -0.673690 4.0
5 -0.879758  0.113648 5.0
6 -1.040039 -1.478427 6.0
7 -0.772263  0.524988 7.0

In [9]: df3.dtypes
Out[9]:
A  float32
B  float64
C  float64
dtype: object
Dtype conversion

This is lower-common-denominator upcasting, meaning you get the dtype which can accommodate all of the types

```
In [10]: df3.values.dtype
Out[10]: dtype('float64')
```

Conversion

```
In [11]: df3.astype('float32').dtypes
Out[11]:
A float32
B float32
C float32
dtype: object
```

Mixed conversion

```
In [12]: df3['D'] = '1.'
In [13]: df3['E'] = '1'

In [14]: df3.convert_objects(convert_numeric=True).dtypes
Out[14]:
A float32
B float64
C float64
D float64
E int64
dtype: object
```

Forcing date coercion (and setting NaT when not datelike)

```
In [18]: import datetime
In [19]: s = pd.Series([datetime.datetime(2001, 1, 1, 0, 0), 'foo', 1.0, 1, ....:
                     pd.Timestamp('20010104'), '20010105'], dtype='O')
......:
In [20]: s.convert_objects(convert_dates='coerce')
```

(continues on next page)
Dtype gotchas

Platform gotchas

Starting in 0.11.0, construction of DataFrame/Series will use default dtypes of int64 and float64, regardless of platform. This is not an apparent change from earlier versions of pandas. If you specify dtypes, they WILL be respected, however (GH2837)

The following will all result in int64 dtypes

```
In [21]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[21]:
    a    int64
dtype: object

In [22]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[22]:
    a    int64
dtype: object

In [23]: pd.DataFrame({'a': 1}, index=range(2)).dtypes
Out[23]:
    a    int64
dtype: object
```

Keep in mind that DataFrame(np.array([1,2])) WILL result in int32 on 32-bit platforms!

Upcasting gotchas

Performing indexing operations on integer type data can easily upcast the data. The dtype of the input data will be preserved in cases where nans are not introduced.

```
In [24]: dfi = df3.astype('int32')

In [25]: dfi['D'] = dfi['D'].astype('int64')

In [26]: dfi
Out[26]:
   A  B  C  D  E
0  0  0  0  1  1
1 -2  0  1  1  1
2 -2  0  2  1  1
3  0 -1  3  1  1
4  1  0  4  1  1
```
In [27]: dfi.dtypes
Out[27]:
A   int32
B   int32
C   int32
D   int64
E   int32
dtype: object

In [28]: casted = dfi[dfi > 0]

In [29]: casted
Out[29]:
   A   B   C   D   E
0   NaN NaN  NaN  1.0  1.0
1   NaN NaN  2.0  1.0  1.0
2   NaN NaN  3.0  1.0  1.0
3   1.0 NaN  4.0  1.0  1.0
4   NaN NaN  5.0  1.0  1.0
5   NaN NaN  6.0  1.0  1.0
6   NaN NaN  7.0  1.0  1.0

In [30]: casted.dtypes
Out[30]:
A   float64
B   float64
C   float64
D   int64
E   int32
dtype: object

While float dtypes are unchanged.

In [31]: df4 = df3.copy()

In [32]: df4['A'] = df4['A'].astype('float32')

In [33]: df4.dtypes
Out[33]:
A   float32
B   float64
C   float64
D   float16
E   int32
dtype: object

In [34]: casted = df4[df4 > 0]
In [35]: casted
Out[35]:
    A    B    C    D    E
0   NaN   NaN   NaN  1.0   1
1   NaN  0.567020   1.0   1.0   1
2   NaN  0.276232   2.0   1.0   1
3   NaN   NaN   3.0   1.0   1
4  1.933792   NaN   4.0   1.0   1
5   NaN  0.113648   5.0   1.0   1
6   NaN   NaN   6.0   1.0   1
7   NaN  0.524988   7.0   1.0   1

In [36]: casted.dtypes
Out[36]:
A float32
B float64
C float64
D float16
E int32
dtype: object

Datetimes conversion

Datetime64[ns] columns in a DataFrame (or a Series) allow the use of np.nan to indicate a nan value, in addition to the traditional NaT, or not-a-time. This allows convenient nan setting in a generic way. Furthermore datetime64[ns] columns are created by default, when passed datetimelike objects (this change was introduced in 0.10.1) (GH2809, GH2810)

In [12]: df = pd.DataFrame(np.random.randn(6, 2), pd.date_range('20010102', periods=6),
...:                     columns=['A', 'B'])

In [13]: df['timestamp'] = pd.Timestamp('20010103')

In [14]: df
Out[14]:
   A       B  timestamp
0  0.404705  0.577046 2001-01-02
1 -1.715002 -1.039268 2001-01-02
2 -0.370647 -1.157892 2001-01-02
3 -1.344312  0.844885 2001-01-02
4  1.075770 -0.109050 2001-01-02
5  1.643563 -1.469388 2001-01-02

# datetime64[ns] out of the box
In [15]: df.dtypes.value_counts()
Out[15]:
float64    2
datetime64[ns]    1
dtype: int64
# use the traditional nan, which is mapped to NaT internally
In [16]: df.loc[df.index[2:4], ['A', 'timestamp']] = np.nan

In [17]: df
Out[17]:
   A      B    timestamp
0  0.404705  0.577046  2001-01-02
1 -1.715002 -1.039268  2001-01-03
2  NaN    -1.157892  NaT
3  NaN     0.844885  NaT
4  1.075770 -0.109050  2001-01-03
5  1.643563 -1.469388  2001-01-07

Astype conversion on datetime64[ns] to object, implicitly converts NaT to np.nan

In [18]: import datetime

In [19]: s = pd.Series([datetime.datetime(2001, 1, 2, 0, 0) for i in range(3)])

In [20]: s
Out[20]:
0  2001-01-02
1  NaT
2  2001-01-02
dtype: datetime64[ns]

In [21]: s[1] = np.nan

In [22]: s
Out[22]:
0  2001-01-02
1  NaT
2  2001-01-02
dtype: datetime64[ns]

In [23]: s.dtype
Out[23]: dtype('<M8[ns]')

In [24]: s = s.astype('O')

In [25]: s
Out[25]:
0  2001-01-02 00:00:00
1  NaT
2  2001-01-02 00:00:00
dtype: object

In [26]: s.dtype
Out[26]: dtype('O')
API changes

- Added to_series() method to indices, to facilitate the creation of indexers (GH3275)
- HDFStore
  - added the method select_column to select a single column from a table as a Series.
  - deprecated the unique method, can be replicated by select_column(key, column).unique()
  - min_itemsize parameter to append will now automatically create data_columns for passed keys

Enhancements

- Improved performance of df.to_csv() by up to 10x in some cases. (GH3059)
- Numexpr is now a Recommended Dependencies, to accelerate certain types of numerical and boolean operations
- Bottleneck is now a Recommended Dependencies, to accelerate certain types of nan operations
- HDFStore
  - support read_hdf/to_hdf API similar to read_csv/to_csv

```
In [27]: df = pd.DataFrame({"A": range(5), "B": range(5)})
In [28]: df.to_hdf("store.h5", "table", append=True)
In [29]: pd.read_hdf("store.h5", "table", where=["index > 2"])
Out[29]:
   A  B
  3  3  3
  4  4  4
```

- provide dotted attribute access to get from stores, e.g. store.df == store['df']
- new keywords iterator=boolean, and chunksize=number_in_a_chunk are provided to support iteration on select and select_as_multiple (GH3076)

- You can now select timestamps from an unordered timeseries similarly to an ordered timeseries (GH2437)
- You can now select with a string from a DataFrame with a datelike index, in a similar way to a Series (GH3070)

```
In [30]: idx = pd.date_range("2001-10-1", periods=5, freq='M')
In [31]: ts = pd.Series(np.random.rand(len(idx)), index=idx)
In [32]: ts["2001"]
Out[32]:
2001-10-31  0.117967
2001-11-30  0.702184
2001-12-31  0.414034
Freq: M, dtype: float64
In [33]: df = pd.DataFrame({"A": ts})
In [34]: df["2001"]
Out[34]:
```

(continues on next page)
• Squeeze to possibly remove length 1 dimensions from an object.

```python
>>> p = pd.Panel(np.random.randn(3, 4, 4), items=['ItemA', 'ItemB', 'ItemC'],
               major_axis=pd.date_range('20010102', periods=4),
               minor_axis=['A', 'B', 'C', 'D'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D

>>> p.reindex(items=['ItemA']).squeeze()
A    B    C    D
2001-01-02  0.926089 -2.026458  0.501277 -0.204683
2001-01-03 -0.076524  1.081161  1.141361  0.479243
2001-01-04  0.641817 -0.185352  1.824568  0.809152
2001-01-05  0.575237  0.669934  1.398014 -0.399338
Freq: D, Name: B, dtype: float64

>>> p.reindex(items=['ItemA'], minor=['B']).squeeze()
2001-01-02 -2.026458
2001-01-03  1.081161
2001-01-04 -0.185352
2001-01-05  0.669934
Freq: D, Name: B, dtype: float64
```

• In `pd.io.data.Options`,
  - Fix bug when trying to fetch data for the current month when already past expiry.
  - Now using lxml to scrape html instead of BeautifulSoup (lxml was faster).
  - New instance variables for calls and puts are automatically created when a method that creates them is called. This works for current month where the instance variables are simply `calls` and `puts`. Also works for future expiry months and save the instance variable as `callsMMYY` or `putsMMYY`, where `MMYY` are, respectively, the month and year of the option’s expiry.
  - `Options.get_near_stock_price` now allows the user to specify the month for which to get relevant options data.
  - `Options.get_forward_data` now has optional kwargs `near` and `above_below`. This allows the user to specify if they would like to only return forward looking data for options near the current stock price. This just obtains the data from `Options.get_near_stock_price` instead of `Options.get_xxx_data()` (GH2758).

• Cursor coordinate information is now displayed in time-series plots.

• added option `display.max_seq_items` to control the number of elements printed per sequence pprinting it. (GH2979)

• added option `display.chop_threshold` to control display of small numerical values. (GH2739)
• added option `display.max_info_rows` to prevent `verbose_info` from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)

• `value_counts()` now accepts a “normalize” argument, for normalized histograms. (GH2710).

• `DataFrame.from_records` now accepts not only dicts but any instance of the collections.Mapping ABC.

• added option `display.mpl_style` providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).

• Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)

• `to_html()` now accepts an optional “escape” argument to control reserved HTML character escaping (enabled by default) and escapes &, in addition to < and >. (GH2919)

See the full release notes or issue tracker on GitHub for a complete list.

Contributors

A total of 50 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Adam Greenhall +
• Alvaro Tejero-Cantero +
• Andy Hayden
• Brad Buran +
• Chang She
• Chapman Siu +
• Chris Withers +
• Christian Geier +
• Christopher Whelan
• Damien Garaud
• Dan Birken
• Dan Davison +
• Dieter Vandenbussche
• Drazen Lucanin +
• Dražen Lučanin +
• Garrett Drapala
• Illia Polosukhin +
• James Casbon +
• Jeff Reback
• Jeremy Wagner +
• Jonathan Chambers +
• K.-Michael Aye
• Karmel Allison +
5.21 Version 0.10

5.21.1 Version 0.10.1 (January 22, 2013)

This is a minor release from 0.10.0 and includes new features, enhancements, and bug fixes. In particular, there is substantial new HDFStore functionality contributed by Jeff Reback.

An undesired API breakage with functions taking the \texttt{inplace} option has been reverted and deprecation warnings added.
**API changes**

- Functions taking an `inplace` option return the calling object as before. A deprecation message has been added
- Groupby aggregations Max/Min no longer exclude non-numeric data (GH2700)
- Resampling an empty DataFrame now returns an empty DataFrame instead of raising an exception (GH2640)
- The file reader will now raise an exception when NA values are found in an explicitly specified integer column instead of converting the column to float (GH2631)
- `DatetimeIndex.unique` now returns a `DatetimeIndex` with the same name and timezone instead of an array (GH2563)

**New features**

- MySQL support for database (contribution from Dan Allan)

**HDFStore**

You may need to upgrade your existing data files. Please visit the [compatibility](https://pandas.pydata.org/docs/user_guide/compatibility.html) section in the main docs.

You can designate (and index) certain columns that you want to be able to perform queries on a table, by passing a list to `data_columns`.

```python
In [1]: store = pd.HDFStore("store.h5")

In [2]: df = pd.DataFrame(
    ...:     np.random.randn(8, 3),
    ...:     index=pd.date_range("1/1/2000", periods=8),
    ...:     columns=["A", "B", "C"],
    ...:     )

In [3]: df["string"] = "foo"

In [4]: df.loc[df.index[4:6], "string"] = np.nan

In [5]: df.loc[df.index[7:9], "string"] = "bar"

In [6]: df["string2"] = "cool"

In [7]: df
```

```
Out[7]:
A     B     C     string  string2
2000-01-01 0.469112 -0.282863 -1.509059    foo       cool
2000-01-02 -1.135632  1.212112  -0.173215    foo       cool
2000-01-03  0.119209 -1.044236  -0.861849    foo       cool
2000-01-04 -2.104569  0.494929   1.071804    foo       cool
2000-01-05  0.721555  0.706771  -1.039575     NaN      cool
2000-01-06  0.271860  0.424972   0.567020     NaN      cool
2000-01-07  0.276232 -1.087401  -0.673690    foo       cool
2000-01-08  0.113648 -1.478427   0.524988     bar       cool
```

(continues on next page)
# on-disk operations
In [8]: store.append("df", df, data_columns=["B", "C", "string", "string2"])  

In [9]: store.select("df", "B>0 and string=='foo'")  
Out[9]:  2000-01-02 -1.135632 1.212112 -0.173215 foo cool  

# this is in-memory version of this type of selection  
In [10]: df[(df.B > 0) & (df.string == "foo")]
Out[10]:  2000-01-02 -1.135632 1.212112 -0.173215 foo cool  

Retrieving unique values in an indexable or data column.

# note that this is deprecated as of 0.14.0  
# can be replicated by: store.select_column('df','index').unique()  
store.unique("df", "index")  
store.unique("df", "string")  

You can now store datetime64 in data columns

In [11]: df_mixed = df.copy()  
In [12]: df_mixed["datetime64"] = pd.Timestamp("20010102")  
In [13]: df_mixed.loc[df_mixed.index[3:4], ["A", "B"]]= np.nan  
In [14]: store.append("df_mixed", df_mixed)  
In [15]: df_mixed1 = store.select("df_mixed")  

In [16]: df_mixed1
Out[16]:  A  B  C  string  string2  datetime64  
2000-01-01 0.469112 -0.282863 -1.509059 foo cool 2001-01-02  
2000-01-02 -1.135632 1.212112 -0.173215 foo cool 2001-01-02  
2000-01-03 0.119209 -1.044236 -0.861849 foo cool 2001-01-02  
2000-01-04 NaN NaN 1.071804 foo cool 2001-01-02  
2000-01-05 0.721555 -0.706771 -1.039575 NaN cool 2001-01-02  
2000-01-06 0.271860 -0.424972 0.567020 NaN cool 2001-01-02  
2000-01-07 0.276232 -1.087401 -0.673690 foo cool 2001-01-02  
2000-01-08 0.113648 -1.478427 0.524988 bar cool 2001-01-02  

In [17]: df_mixed1.dtypes.value_counts()
Out[17]:  float64 3  
object 2  
datetime64[ns] 1  
dtype: int64  

You can pass columns keyword to select to filter a list of the return columns, this is equivalent to passing a
Term('columns',list_of_columns_to_filter)

```python
In [18]: store.select("df", columns=['A', 'B'])
Out[18]:
     A      B
2000-01-01  0.469112 -0.282863
2000-01-02 -1.135632  1.212112
2000-01-03  0.119209 -1.044236
2000-01-04 -2.104569 -0.494929
2000-01-05  0.721555  0.706771
2000-01-06  0.271860 -0.424972
2000-01-07  0.276232 -1.087401
2000-01-08  0.113648 -1.478427
```

HDFStore now serializes MultiIndex dataframes when appending tables.

```python
In [19]: index = pd.MultiIndex(levels=[[
'foo', 'bar', 'baz', 'qux'],
   ['one', 'two', 'three']],
   labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
          [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   names=['foo', 'bar'])

In [20]: df = pd.DataFrame(np.random.randn(10, 3), index=index,
   columns=['A', 'B', 'C'])

In [21]: df
Out[21]:
     A      B      C
foo bar
  one -0.116619  0.295575 -1.047704
  two  1.640556  1.905836  2.772115
  three  0.088787 -1.144197  0.633372
bar one  0.925372 -0.006438 -0.820408
  two -0.600874 -1.039266  0.824758
baz two -0.824095 -0.337730  0.927764
  three -0.840123  0.248505  0.109250
qux one  0.431977 -0.460710  0.336505
  two -3.207595 -1.535854  0.409769
  three  0.673145 -0.741113 -0.110891
```

```python
In [22]: store.append('mi', df)
In [23]: store.select('mi')
Out[23]:
     A      B      C
foo bar
  one -0.116619  0.295575 -1.047704
  two  1.640556  1.905836  2.772115
  three  0.088787 -1.144197  0.633372
bar one  0.925372 -0.006438 -0.820408
  two -0.600874 -1.039266  0.824758
baz two -0.824095 -0.337730  0.927764
  three -0.840123  0.248505  0.109250
```

(continues on next page)
three -0.840123  0.248505 -0.109250
qux one  0.431977 -0.460710  0.336505
two  -3.207595 -1.535854  0.409769
three -0.673145 -0.741113  -0.110891

# the levels are automatically included as data columns
In [24]: store.select("mi", "foo='bar'")
Out[24]:
A    B    C
foo bar
   bar one  0.925372 -0.006438 -0.820408
two -0.600874 -1.039266  0.824758

Multi-table creation via append_to_multiple and selection via select_as_multiple can create/select from multiple tables and return a combined result, by using where on a selector table.

In [19]: df_mt = pd.DataFrame(
    ....:     np.random.randn(8, 6),
    ....:     index=pd.date_range("1/1/2000", periods=8),
    ....:     columns=["A", "B", "C", "D", "E", "F"],
    ....:     )
    ....:
In [20]: df_mt["foo"] = "bar"

# you can also create the tables individually
In [21]: store.append_to_multiple(
    ....:     {"df1_mt": ["A", "B"], "df2_mt": None}, df_mt, selector="df1_mt"
    ....:     )
    ....:
In [22]: store
Out[22]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [23]: store.select("df1_mt")
Out[23]:
   A    B
2000-01-01  0.404705  0.577046
2000-01-02 -1.344312  0.844885
2000-01-03  0.357021 -0.674600
2000-01-04  0.276662 -0.472035
2000-01-05  0.895717  0.805244
2000-01-06 -1.170299 -0.226169
2000-01-07 -0.076467 -1.187678
2000-01-08  1.024180  0.569605

In [24]: store.select("df2_mt")
Out[24]:
   C    D    E    F  foo
2000-01-01 -1.715002 -1.039268 -0.370647 -1.157892 bar
2000-01-02  1.075770 -0.109050  1.643563 -1.469388 bar
2000-01-03 -1.776904 -0.968914 -1.294524  0.413738 bar
2000-01-04 -0.013960 -0.362543 -0.006154 -0.923061 bar
2000-01-05 -1.206412  2.565646  1.431256  1.340309 bar
2000-01-06  0.410835  0.813850  0.132003 -0.827317 bar
2000-01-07  1.130127 -1.436737 -1.413681  1.607920 bar
2000-01-08  0.875906 -2.211372  0.974466 -2.006747 bar

# as a multiple
In [25]: store.select_as_multiple(
   ....:     ["df1_mt", "df2_mt"], where=["A>0", "B>0"], selector="df1_mt"
   ....: )
   ....:
Out[25]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
<td>-1.039268</td>
<td>-0.370647</td>
<td>-1.157892</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.895717</td>
<td>0.805244</td>
<td>-1.206412</td>
<td>2.565646</td>
<td>1.431256</td>
<td>1.340309</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>1.024180</td>
<td>0.569605</td>
<td>0.875906</td>
<td>-2.211372</td>
<td>0.974466</td>
<td>-2.006747</td>
</tr>
</tbody>
</table>

Enhancements

- **HDFStore** now can read native PyTables table format tables
- You can pass `nan_rep = 'my_nan_rep'` to append, to change the default nan representation on disk (which converts to/from np.nan), this defaults to nan.
- You can pass `index` to append. This defaults to True. This will automagically create indices on the `indexables` and data columns of the table
- You can pass `chunksize=an integer` to append, to change the writing chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=an integer` to the first `append`, to set the TOTAL number of expected rows that PyTables will expected. This will optimize read/write performance.
- `Select` now supports passing `start` and `stop` to provide selection space limiting in selection.
- Greatly improved ISO8601 (e.g., yyyy-mm-dd) date parsing for file parsers (GH2698)
- Allow `DataFrame.merge` to handle combinatorial sizes too large for 64-bit integer (GH2690)
- Series now has unary negation (negation) and inversion (neg) operators (GH2686)
- `DataFrame.plot` now includes a `logx` parameter to change the x-axis to log scale (GH2327)
- Series arithmetic operators can now handle constant and ndarray input (GH2574)
- ExcelFile now takes a kind argument to specify the file type (GH2613)
- A faster implementation for Series.str methods (GH2602)

Bug Fixes

- **HDFStore** tables can now store `float32` types correctly (cannot be mixed with `float64` however)
- Fixed Google Analytics prefix when specifying request segment (GH2713).
- Function to reset Google Analytics token store so users can recover from improperly setup client secrets (GH2687).
- Fixed groupby bug resulting in segfault when passing in MultiIndex (GH2706)
• Fixed bug where passing a Series with datetime64 values into `to_datetime` results in bogus output values (GH2699)
• Fixed bug in `pattern` in HDFStore expressions when pattern is not a valid regex (GH2694)
• Fixed performance issues while aggregating boolean data (GH2692)
• When given a boolean mask key and a Series of new values, Series `__setitem__` will now align the incoming values with the original Series (GH2686)
• Fixed MemoryError caused by performing counting sort on sorting MultiIndex levels with a very large number of combinatorial values (GH2684)
• Fixed bug that causes plotting to fail when the index is a DatetimeIndex with a fixed-offset timezone (GH2683)
• Corrected business day subtraction logic when the offset is more than 5 bdays and the starting date is on a weekend (GH2680)
• Fixed C file parser behavior when the file has more columns than data (GH2668)
• Fixed file reader bug that misaligned columns with data in the presence of an implicit column and a specified `usecols` value
• DataFrames with numerical or datetime indices are now sorted prior to plotting (GH2609)
• Fixed DataFrame.from_records error when passed columns, index, but empty records (GH2633)
• Several bug fixed for Series operations when dtype is datetime64 (GH2689, GH2629, GH2626)

See the full release notes or issue tracker on GitHub for a complete list.

Contributors

A total of 17 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Andy Hayden +
• Anton I. Sipos +
• Chang She
• Christopher Whelan
• Damien Garaud +
• Dan Allan +
• Dieter Vandenbussche
• Garrett Drapala +
• Jay Parlar +
• Thouis (Ray) Jones +
• Vincent Arel-Bundock +
• Wes McKinney
• elpres
• herrfz +
• jreback
• svaksha +
5.21.2 Version 0.10.0 (December 17, 2012)

This is a major release from 0.9.1 and includes many new features and enhancements along with a large number of bug fixes. There are also a number of important API changes that long-time pandas users should pay close attention to.

File parsing new features

The delimited file parsing engine (the guts of `read_csv` and `read_table`) has been rewritten from the ground up and now uses a fraction the amount of memory while parsing, while being 40% or more faster in most use cases (in some cases much faster).

There are also many new features:

- Much-improved Unicode handling via the `encoding` option.
- Column filtering (`usecols`)
- Dtype specification (`dtype` argument)
- Ability to specify strings to be recognized as True/False
- Ability to yield NumPy record arrays (`as_recarray`)
- High performance `delim_whitespace` option
- Decimal format (e.g. European format) specification
- Easier CSV dialect options: `escapechar`, `lineterminator`, `quotechar`, etc.
- More robust handling of many exceptional kinds of files observed in the wild

API changes

Deprecated DataFrame BINOP TimeSeries special case behavior

The default behavior of binary operations between a DataFrame and a Series has always been to align on the DataFrame’s columns and broadcast down the rows, except in the special case that the DataFrame contains time series. Since there are now method for each binary operator enabling you to specify how you want to broadcast, we are phasing out this special case (Zen of Python: Special cases aren’t special enough to break the rules). Here’s what I’m talking about:

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(np.random.randn(6, 4), index=pd.date_range("1/1/2000", period=6))

In [3]: df
Out[3]:
     0      1      2      3
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690 -0.113648 -1.478427  0.524988
```

(continues on next page)
# deprecated now

In [4]: df - df[0]

Out[4]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>00:00:00</td>
<td>2000-01-02</td>
<td>00:00:00</td>
<td>2000-01-03</td>
</tr>
<tr>
<td>...00:00 ..</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2000-01-01</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-03</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-05</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-01-06</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...NaN ... NaN NaN NaN NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[6 rows x 10 columns]

# Change your code to

In [5]: df.sub(df[0], axis=0)  

# align on axis 0 (rows)

Out[5]:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.0</td>
<td>-0.751976</td>
<td>-1.978171</td>
<td>-1.604745</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.0</td>
<td>-1.385327</td>
<td>-1.092903</td>
<td>-2.256348</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>0.0</td>
<td>-1.242720</td>
<td>0.366920</td>
<td>1.933653</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.0</td>
<td>-1.428326</td>
<td>-1.761130</td>
<td>-0.449695</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.0</td>
<td>0.991993</td>
<td>0.701204</td>
<td>-0.662428</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.0</td>
<td>0.787338</td>
<td>-0.804737</td>
<td>1.198677</td>
</tr>
</tbody>
</table>

You will get a deprecation warning in the 0.10.x series, and the deprecated functionality will be removed in 0.11 or later.

Altered resample default behavior

The default time series resample binning behavior of daily D and higher frequencies has been changed to closed='left', label='left'. Lower frequencies are unaffected. The prior defaults were causing a great deal of confusion for users, especially resampling data to daily frequency (which labeled the aggregated group with the end of the interval: the next day).

In [1]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')

In [2]: series = pd.Series(np.arange(len(dates)), index=dates)

In [3]: series

Out[3]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>00:00:00</td>
<td>2000-01-01</td>
<td>04:00:00</td>
<td>2000-01-01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
pandas: powerful Python data analysis toolkit, Release 1.4.2

(continued from previous page)

2000-01-01 20:00:00 5
2000-01-02 00:00:00 6
2000-01-02 04:00:00 7
2000-01-02 08:00:00 8
2000-01-02 12:00:00 9
2000-01-02 16:00:00 10
2000-01-02 20:00:00 11
2000-01-03 00:00:00 12
2000-01-03 04:00:00 13
2000-01-03 08:00:00 14
2000-01-03 12:00:00 15
2000-01-03 16:00:00 16
2000-01-03 20:00:00 17
2000-01-04 00:00:00 18
2000-01-04 04:00:00 19
2000-01-04 08:00:00 20
2000-01-04 12:00:00 21
2000-01-04 16:00:00 22
2000-01-04 20:00:00 23
2000-01-05 00:00:00 24
Freq: 4H, dtype: int64

In [4]: series.resample('D', how='sum')
Out[4]:
2000-01-01 15
2000-01-02 51
2000-01-03 87
2000-01-04 123
2000-01-05 24
Freq: D, dtype: int64

In [5]: # old behavior
In [6]: series.resample('D', how='sum', closed='right', label='right')
Out[6]:
2000-01-01 0
2000-01-02 21
2000-01-03 57
2000-01-04 93
2000-01-05 129
Freq: D, dtype: int64

• Infinity and negative infinity are no longer treated as NA by isnull and notnull. That they ever were was a relic of early pandas. This behavior can be re-enabled globally by the mode.use_inf_as_null option:

In [6]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])
In [7]: pd.isnull(s)
Out[7]:
0   False
1   False
2   False
3   False

(continues on next page)
Length: 4, dtype: bool

In [8]: s.fillna(0)
Out[8]:
0  1.500000
1   inf
2  3.400000
3   -inf
Length: 4, dtype: float64

In [9]: pd.set_option('use_inf_as_null', True)

In [10]: pd.isnull(s)
Out[10]:
0   False
1    True
2   False
3    True
Length: 4, dtype: bool

In [11]: s.fillna(0)
Out[11]:
0   1.5
1    0.0
2   3.4
3    0.0
Length: 4, dtype: float64

In [12]: pd.reset_option('use_inf_as_null')

- Methods with the inplace option now all return None instead of the calling object. E.g. code written like df =
df.fillna(0, inplace=True) may stop working. To fix, simply delete the unnecessary variable assignment.

- pandas.merge no longer sorts the group keys (sort=False) by default. This was done for performance reasons:
the group-key sorting is often one of the more expensive parts of the computation and is often unnecessary.

- The default column names for a file with no header have been changed to the integers 0 through n - 1. This is
to create consistency with the DataFrame constructor with no columns specified. The v0.9.0 behavior (names
X0, X1, ...) can be reproduced by specifying prefix='X':

In [6]: import io

In [7]: data = """
...: a,b,c
...: 1,Yes,2
...: 3,No,4
...: """
...: print(data)
...: pd.read_csv(io.StringIO(data), header=None)
...: pd.read_csv(io.StringIO(data), header=None, prefix="X")

Input In [7]
print(data)
• Values like 'Yes' and 'No' are not interpreted as boolean by default, though this can be controlled by new
true_values and false_values arguments:

In [4]: print(data)
   a, b, c
   1, Yes, 2
   3, No, 4

In [5]: pd.read_csv(io.StringIO(data))
Out[5]:
   a  b  c
  0  1  Yes  2
  1  3   No  4

In [6]: pd.read_csv(io.StringIO(data), true_values=["Yes"], false_values=["No"],)
Out[6]:
   a  b  c
  0  1  True  2
  1  3  False  4

• The file parsers will not recognize non-string values arising from a converter function as NA if passed in the
na_values argument. It’s better to do post-processing using the replace function instead.

• Calling fillna on Series or DataFrame with no arguments is no longer valid code. You must either specify a
fill value or an interpolation method:

In [8]: s = pd.Series([np.nan, 1.0, 2.0, np.nan, 4])

In [9]: s
Out[9]:
   0   NaN
   1  1.0
   2  2.0
   3   NaN
   4  4.0
dtype: float64

In [10]: s.fillna(0)
Out[10]:
   0   0.0
   1  1.0
   2  2.0
   3   0.0
   4  4.0
dtype: float64

In [11]: s.fillna(method="pad")
Out[11]:
   0   NaN
Convenience methods \texttt{ffill} and \texttt{bfill} have been added:

\begin{verbatim}
In [12]: s.ffill()
Out[12]:
0    NaN
1    1.0
2    2.0
3    2.0
4    4.0
dtype: float64
\end{verbatim}

- \texttt{Series.apply} will now operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame

\begin{verbatim}
In [13]: def f(x):
    ....:     return pd.Series([x, x ** 2], index=["x", "x^2"])
    ....:

In [14]: s = pd.Series(np.random.rand(5))

In [15]: s
Out[15]:
0    0.340445
1    0.984729
2    0.919540
3    0.037772
4    0.861549
dtype: float64

In [16]: s.apply(f)
Out[16]:
   x  x^2
0  0.340445  0.115903
1  0.984729  0.969691
2  0.919540  0.845555
3  0.037772  0.001427
4  0.861549  0.742267
\end{verbatim}

- New API functions for working with pandas options (GH2097):
  - \texttt{get_option} / \texttt{set_option} - get/set the value of an option. Partial names are accepted. - \texttt{reset_option} - reset one or more options to their default value. Partial names are accepted. - \texttt{describe_option} - print a description of one or more options. When called with no arguments. print all registered options.

Note: \texttt{set_printoptions}/\texttt{reset_printoptions} are now deprecated (but functioning), the print options now live under "display.XYZ". For example:
New features

Wide DataFrame printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```python
In [18]: wide_frame = pd.DataFrame(np.random.randn(5, 16))
```

```plaintext
0 1 2 3 4 5 ... 10 11
0 -0.548702 1.467327 -1.015962 -0.483075 1.637550 ... -0.919069 0.266046 -0.291519 -1.745505 -0.483075
-0.709661 1.669052 1.037882 -1.705775 1.037882 -1.705775
-0.919854 -0.423797 1.247642 -0.009920 0.290213 0.495767 ... 0.337863 -0.945867 -0.945867
-0.932132 1.956030 0.017587 -0.016692 1.956030 0.017587
-0.575247 0.254161 -1.143704 0.215897 1.193555 -0.077118 ... 1.627081 -0.990582 -0.990582
```

The width of each line can be changed via `line_width` (80 by default):

```python
In [20]: pd.set_option("line_width", 80)
```

The old behavior of printing out summary information can be achieved via the ‘expand_frame_repr’ print option:

```python
In [21]: wide_frame
```

```plaintext
0 1 2 3 4 5 ... 10 11
0 -0.548702 1.467327 -1.015962 -0.483075 1.637550 ... -0.919069 0.266046 -0.291519 -1.745505 -0.483075
-0.709661 1.669052 1.037882 -1.705775 1.037882 -1.705775
-0.919854 -0.423797 1.247642 -0.009920 0.290213 0.495767 ... 0.337863 -0.945867 -0.945867
-0.932132 1.956030 0.017587 -0.016692 1.956030 0.017587
-0.575247 0.254161 -1.143704 0.215897 1.193555 -0.077118 ... 1.627081 -0.990582 -0.990582
```

The width of each line can be changed via ‘line_width’ (80 by default):
Updated PyTables support

*Docs* for PyTables Table format & several enhancements to the api. Here is a taste of what to expect.

```python
In [41]: store = pd.HDFStore('store.h5')
In [42]: df = pd.DataFrame(np.random.randn(8, 3),
                       index=pd.date_range('1/1/2000', periods=8),
                       columns=['A', 'B', 'C'])
In [43]: df
Out[43]:
     A         B         C
2000-01-01 -2.036047  0.000830 -0.955697
2000-01-02 -0.898872 -0.725411  0.059904
2000-01-03 -0.449644  1.082900 -1.221265
2000-01-04  0.361078  1.330900  0.855932
2000-01-05 -1.216718  1.488887  0.018993
2000-01-06 -0.877046  0.045976  0.437274
2000-01-07 -0.567182 -0.888657 -0.556383
2000-01-08  0.655457  1.117949 -2.782376
```

# appending data frames

```python
In [44]: df1 = df[0:4]
In [45]: df2 = df[4:]
In [46]: store.append('df', df1)
In [47]: store.append('df', df2)
In [48]: store
Out[48]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df          frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
```

# selecting the entire store

```python
In [49]: store.select('df')
Out[49]:
     A         B         C
2000-01-01 -2.036047  0.000830 -0.955697
2000-01-02 -0.898872 -0.725411  0.059904
2000-01-03 -0.449644  1.082900 -1.221265
2000-01-04  0.361078  1.330900  0.855932
2000-01-05 -1.216718  1.488887  0.018993
```

(continues on next page)
2000-01-06  -0.877046  0.045976  0.437274
2000-01-07  -0.567182  -0.888657  -0.556383
2000-01-08   0.655457  1.117949  -2.782376

[8 rows x 3 columns]

In [50]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   ....:     major_axis=pd.date_range('1/1/2000', periods=5),
   ....:     minor_axis=['A', 'B', 'C', 'D'])

In [51]: wp
Out[51]: <class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

# storing a panel
In [52]: store.append('wp', wp)

# selecting via A QUERY
In [53]: store.select('wp', [pd.Term('major_axis>20000102'),
   ....:     pd.Term('minor_axis', '=', ['A', 'B'])])
Out[53]: <class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B

# removing data from tables
In [54]: store.remove('wp', pd.Term('major_axis>20000103'))
Out[54]: 8

In [55]: store.select('wp')
Out[55]: <class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to D

# deleting a store
In [56]: del store['df']

In [57]: store
Out[57]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp wide_table (typ->appendable,nrows->12,ncols->2,indexers->[major_axis,
   ....:     minor_axis])
Enhancements

- added ability to hierarchical keys

```python
In [58]: store.put('foo/bar/bah', df)
In [59]: store.append('food/orange', df)
In [60]: store.append('food/apple', df)

In [61]: store
Out[61]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/foo/bar/bah frame (shape->[8,3])
/food/apple frame_table (typ->appendable,nrows->8,ncols->3,
   →indexers->[index])
/food/orange frame_table (typ->appendable,nrows->8,ncols->3,
   →indexers->[index])
/wp wide_table (typ->appendable,nrows->12,ncols->2,
   →indexers->[major_axis,minor_axis])

# remove all nodes under this level
In [62]: store.remove('food')

In [63]: store
Out[63]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/foo/bar/bah frame (shape->[8,3])
/wp wide_table (typ->appendable,nrows->12,ncols->2,
   →indexers->[major_axis,minor_axis])
```

- added mixed-dtype support!

```python
In [64]: df['string'] = 'string'
In [65]: df['int'] = 1
In [66]: store.append('df', df)
In [67]: df1 = store.select('df')

In [68]: df1
Out[68]:
   A       B        C string int
0 2000-01-01 -2.036047 0.000830 string 1
1 2000-01-02 -0.898872 -0.725411 string 1
2 2000-01-03  0.449644  1.082900 string 1
3 2000-01-04  0.361078  1.330704 string 1
4 2000-01-05  1.216718  1.488887 string 1
(continues on next page)
In [69]: df1.get_dtype_counts()
Out[69]:
float64 3
int64 1
object 1
dtype: int64

• performance improvements on table writing
• support for arbitrarily indexed dimensions
• SparseSeries now has a density property (GH2384)
• enable Series.str.strip/strip methods to take an input argument to strip arbitrary characters (GH2411)
• implement value_vars in melt to limit values to certain columns and add melt to pandas namespace (GH2412)

Bug Fixes
• added Term method of specifying where conditions (GH1996).
• del store['df'] now call store.remove('df') for store deletion
• deleting of consecutive rows is much faster than before
• min_itemsize parameter can be specified in table creation to force a minimum size for indexing columns (the previous implementation would set the column size based on the first append)
• indexing support via create_table_index (requires PyTables >= 2.3) (GH698).
• appending on a store would fail if the table was not first created via put
• fixed issue with missing attributes after loading a pickled dataframe (GH2431)
• minor change to select and remove: require a table ONLY if where is also provided (and not None)

Compatibility
0.10 of HDFStore is backwards compatible for reading tables created in a prior version of pandas, however, query terms using the prior (undocumented) methodology are unsupported. You must read in the entire file and write it out using the new format to take advantage of the updates.

N dimensional panels (experimental)

Adding experimental support for Panel4D and factory functions to create n-dimensional named panels. Here is a taste of what to expect.
In [59]: p4d
Out[59]:
<class 'pandas.core.panel.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

See the full release notes or issue tracker on GitHub for a complete list.

Contributors

A total of 26 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- A. Flaxman +
- Abraham Flaxman
- Adam Obeng +
- Brenda Moon +
- Chang She
- Chris Mulligan +
- Dieter Vandenbussche
- Donald Curtis +
- Jay Bourque +
- Jeff Reback +
- Justin C Johnson +
- K.-Michael Aye
- Keith Hughitt +
- Ken Van Haren +
- Laurent Gautier +
- Luke Lee +
- Martin Blais
- Tobias Brandt +
- Wes McKinney
- Wouter Overmeire
- alex arsenovic +
- jreback +
- locojaydev +
5.22 Version 0.9

5.22.1 Version 0.9.1 (November 14, 2012)

This is a bug fix release from 0.9.0 and includes several new features and enhancements along with a large number of bug fixes. The new features include by-column sort order for DataFrame and Series, improved NA handling for the rank method, masking functions for DataFrame, and intraday time-series filtering for DataFrame.

New features

- Series.sort, DataFrame.sort, and DataFrame.sort_index can now be specified in a per-column manner to support multiple sort orders (GH928)

```python
In [2]: df = pd.DataFrame(np.random.randint(0, 2, (6, 3)),
   ...:    columns=['A', 'B', 'C'])

In [3]: df.sort(['A', 'B'], ascending=[1, 0])
```

```
Out[3]:
    A  B  C
0  3  0  1
1  4  0  1
2  2  0  0
3  0  1  0
4  1  1  0
5  5  1  0
```

- DataFrame.rank now supports additional argument values for the na_option parameter so missing values can be assigned either the largest or the smallest rank (GH1508, GH2159)

```python
In [1]: df = pd.DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])


In [3]: df.rank()
```

```
Out[3]:
    A  B  C
0  3.0  2.0  1.0
1  1.0  3.0  2.0
2  NaN NaN NaN
3  NaN NaN NaN
4  NaN NaN NaN
5  2.0  1.0  3.0
```

```python
In [4]: df.rank(na_option='top')
```

```
Out[4]:
```
In [5]: df.rank(na_option='bottom')
Out[5]:
     A  B  C
0  3.0 2.0 1.0
1  1.0 3.0 2.0
2  5.0 5.0 5.0
3  5.0 5.0 5.0
4  5.0 5.0 5.0
5  2.0 1.0 3.0

• DataFrame has new `where` and `mask` methods to select values according to a given boolean mask (GH2109, GH2151)

DataFrame currently supports slicing via a boolean vector the same length as the DataFrame (inside the []). The returned DataFrame has the same number of columns as the original, but is sliced on its index.

In [6]: df = DataFrame(np.random.randn(5, 3), columns = ['A', 'B', 'C'])

---------------------------------------------------------------------------
NameError                       Traceback (most recent call last)
Input In [6], in <cell line: 1>()
----> 1 df = DataFrame(np.random.randn(5, 3), columns = ['A', 'B', 'C'])
NameError: name 'DataFrame' is not defined

In [7]: df
Out[7]:
     A         B         C
0  0.469112 -0.282863 -1.509059
1 -1.135632  1.212112 -0.173215
2  NaN      NaN      NaN
3  NaN      NaN      NaN
4  NaN      NaN      NaN
5  0.271860 -0.424972  0.567020

In [8]: df[df['A'] > 0]
Out[8]:
     A         B         C
0  0.469112 -0.282863 -1.509059
5  0.271860 -0.424972  0.567020

If a DataFrame is sliced with a DataFrame based boolean condition (with the same size as the original DataFrame), then a DataFrame the same size (index and columns) as the original is returned, with elements that do not meet the boolean condition as NaN. This is accomplished via the new method `DataFrame.where`. In addition, `where` takes an optional `other` argument for replacement.
In [9]: df[df>0]
Out[9]:
   A    B    C
0  0.469112  NaN  NaN
1   NaN  1.212112  NaN
2   NaN   NaN  NaN
3   NaN   NaN  NaN
4   NaN   NaN  NaN
5  0.271860  NaN  0.56702

In [10]: df.where(df>0)
Out[10]:
   A    B    C
0  0.469112  NaN  NaN
1   NaN  1.212112  NaN
2   NaN   NaN  NaN
3   NaN   NaN  NaN
4   NaN   NaN  NaN
5  0.271860  NaN  0.56702

In [11]: df.where(df>0,-df)
Out[11]:
   A    B    C
0  0.469112  0.282863  1.509059
1  1.135632  1.212112  0.173215
2   NaN   NaN  NaN
3   NaN   NaN  NaN
4   NaN   NaN  NaN
5  0.271860  0.424972  0.56702

Furthermore, where now aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via .ix (but on the contents rather than the axis labels)

In [12]: df2 = df.copy()

In [13]: df2[ df2[1:4] > 0 ] = 3

In [14]: df2
Out[14]:
   A    B    C
0  0.469112 -0.282863 -1.509059
1 -1.135632  3.000000 -0.173215
2   NaN   NaN  NaN
3   NaN   NaN  NaN
4   NaN   NaN  NaN
5  0.271860 -0.424972  0.56702

DataFrame.mask is the inverse boolean operation of where.

In [15]: df.mask(df<=0)
Out[15]:
   A    B    C
(continues on next page)
• Enable referencing of Excel columns by their column names (GH1936)

```python
In [16]: xl = pd.ExcelFile('data/test.xls')

In [17]: xl.parse('Sheet1', index_col=0, parse_dates=True,
......:       parse_cols='A:D')
......:
Out[17]:
  A   B   C   D
0  0.469112  NaN  NaN
1   NaN  1.212112  NaN
2   NaN   NaN  NaN
3   NaN   NaN  NaN
4   NaN   NaN  NaN
5  0.271860  NaN  0.56702
```

• Added option to disable pandas-style tick locators and formatters using `series.plot(x_compat=True)` or `pandas.plot_params['x_compat'] = True` (GH2205)

• Existing TimeSeries methods `at_time` and `between_time` were added to DataFrame (GH2149)

• DataFrame.dot can now accept ndarrays (GH2042)

• DataFrame.drop now supports non-unique indexes (GH2101)

• Panel.shift now supports negative periods (GH2164)

• DataFrame now support unary ~ operator (GH2110)

**API changes**

• Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

```python
In [1]: prng = pd.period_range('2012Q1', periods=2, freq='Q')

In [2]: s = pd.Series(np.random.randn(len(prng)), prng)

In [4]: s.resample('M')
Out[4]:
2012-01  -1.471992
2012-02   NaN
2012-03   NaN
2012-04  -0.493593
2012-05   NaN
```

(continues on next page)
• Period.end_time now returns the last nanosecond in the time interval (GH2124, GH2125, GH1764)

```
In [18]: p = pd.Period('2012')
In [19]: p.end_time
Out[19]: Timestamp('2012-12-31 23:59:59.999999999')
```

• File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

```
In [20]: import io
In [21]: data = ('A,B,C
    ....: 00001,001,5
    ....: 00002,002,6')
In [22]: pd.read_csv(io.StringIO(data), converters={'A': lambda x: x.strip()})
Out[22]:
     A  B  C
0  00001 1  5
1  00002 2  6
```

See the full release notes or issue tracker on GitHub for a complete list.

**Contributors**

A total of 11 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Brenda Moon +
- Chang She
- Jeff Reback +
- Justin C Johnson +
- K.-Michael Aye
- Martin Blais
- Tobias Brandt +
- Wes McKinney
- Wouter Overmeire
- timmie
- y-p
5.22.2 Version 0.9.0 (October 7, 2012)

This is a major release from 0.8.1 and includes several new features and enhancements along with a large number of bug fixes. New features include vectorized unicode encoding/decoding for Series.str.to_latex method to DataFrame, more flexible parsing of boolean values, and enabling the download of options data from Yahoo! Finance.

New features

- Add encode and decode for unicode handling to vectorized string processing methods in Series.str (GH1706)
- Add DataFrame.to_latex method (GH1735)
- Add convenient expanding window equivalents of all rolling_* ops (GH1785)
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance (GH1748, GH1739)
- More flexible parsing of boolean values (Yes, No, TRUE, FALSE, etc) (GH1691, GH1295)
- Add level parameter to Series.reset_index
- TimeSeries.between_time can now select times across midnight (GH1871)
- Series constructor can now handle generator as input (GH1679)
- DataFrame.dropna can now take multiple axes (tuple/list) as input (GH924)
- Enable skip_footer parameter in ExcelFile.parse (GH1843)

API changes

- The default column names when header=None and no columns names passed to functions like read_csv has changed to be more Pythonic and amenable to attribute access:

```python
In [1]: import io
In [2]: data = ""
   ...: 0,0,1
   ...: 1,1,0
   ...: 0,1,0
   ...: ""
   ...
In [3]: df = pd.read_csv(io.StringIO(data), header=None)
In [4]: df
Out[4]:
     0  1  2
0  0  0  1
1  1  1  0
2  0  1  0
```

- Creating a Series from another Series, passing an index, will cause reindexing to happen inside rather than treating the Series like an ndarray. Technically improper usages like Series(df[col1], index=df[col2]) that worked before “by accident” (this was never intended) will lead to all NA Series in some cases. To be perfectly clear:
In [5]: s1 = pd.Series([1, 2, 3])
In [6]: s1
Out[6]:
0   1
1   2
2   3
dtype: int64
In [7]: s2 = pd.Series(s1, index=['foo', 'bar', 'baz'])
In [8]: s2
Out[8]:
foo   NaN
bar   NaN
baz   NaN
dtype: float64

- Deprecated day_of_year API removed from PeriodIndex, use dayofyear (GH1723)
- Don’t modify NumPy suppress printoption to True at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by HDFStore (GH1834, GH1824)
- Legacy cruft removed: pandas.stats.misc.quantileTS
- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)
- Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)
- Setting parts of DataFrame/Panel using ix now aligns input Series/DataFrame (GH1630)
- first and last methods in GroupBy no longer drop non-numeric columns (GH1809)
- Resolved inconsistencies in specifying custom NA values in text parser. na_values of type dict no longer override default NAs unless keep_default_na is set to false explicitly (GH1657)
- DataFrame.dot will not do data alignment, and also work with Series (GH1915)

See the full release notes or issue tracker on GitHub for a complete list.

Contributors

A total of 24 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Chang She
- Christopher Whelan +
- Dan Miller +
- Daniel Shapiro +
- Dieter Vandenbussche
- Doug Coleman +
5.23 Version 0.8

5.23.1 Version 0.8.1 (July 22, 2012)

This release includes a few new features, performance enhancements, and over 30 bug fixes from 0.8.0. New features include notably NA friendly string processing functionality and a series of new plot types and options.

New features

- Add vectorized string processing methods accessible via Series.str (GH620)
- Add option to disable adjustment in EWMA (GH1584)
- Radviz plot (GH1566)
- Parallel coordinates plot
- Bootstrap plot
- Per column styles and secondary y-axis plotting (GH1559)
- New datetime converters millisecond plotting (GH1599)
- Add option to disable “sparse” display of hierarchical indexes (GH1538)
- Series/DataFrame’s set_index method can append levels to an existing Index/MultiIndex (GH1569, GH1577)
Performance improvements

- Improved implementation of rolling min and max (thanks to Bottleneck !)
- Add accelerated 'median' GroupBy option (GH1358)
- Significantly improve the performance of parsing ISO8601-format date strings with DatetimeIndex or to_datetime (GH1571)
- Improve the performance of GroupBy on single-key aggregations and use with Categorical types
- Significant datetime parsing performance improvements

Contributors

A total of 5 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Chang She
- Skipper Seabold
- Todd DeLuca +
- Vytautas Jancauskas
- Wes McKinney

5.23.2 Version 0.8.0 (June 29, 2012)

This is a major release from 0.7.3 and includes extensive work on the time series handling and processing infrastructure as well as a great deal of new functionality throughout the library. It includes over 700 commits from more than 20 distinct authors. Most pandas 0.7.3 and earlier users should not experience any issues upgrading, but due to the migration to the NumPy datetime64 dtype, there may be a number of bugs and incompatibilities lurking. Lingering incompatibilities will be fixed ASAP in a 0.8.1 release if necessary. See the full release notes or issue tracker on GitHub for a complete list.

Support for non-unique indexes

All objects can now work with non-unique indexes. Data alignment / join operations work according to SQL join semantics (including, if application, index duplication in many-to-many joins)

NumPy datetime64 dtype and 1.6 dependency

Time series data are now represented using NumPy’s datetime64 dtype, thus, pandas 0.8.0 now requires at least NumPy 1.6. It has been tested and verified to work with the development version (1.7+) of NumPy as well which includes some significant user-facing API changes. NumPy 1.6 also has a number of bugs having to do with nanosecond resolution data, so I recommend that you steer clear of NumPy 1.6’s datetime64 API functions (though limited as they are) and only interact with this data using the interface that pandas provides.

See the end of the 0.8.0 section for a “porting” guide listing potential issues for users migrating legacy code bases from pandas 0.7 or earlier to 0.8.0.

Bug fixes to the 0.7.x series for legacy NumPy < 1.6 users will be provided as they arise. There will be no more further development in 0.7.x beyond bug fixes.
Time Series changes and improvements

Note: With this release, legacy scikits.timeseries users should be able to port their code to use pandas.

Note: See documentation for overview of pandas timeseries API.

- New datetime64 representation speeds up join operations and data alignment, reduces memory usage, and improve serialization / deserialization performance significantly over datetime.datetime

- High performance and flexible resample method for converting from high-to-low and low-to-high frequency. Supports interpolation, user-defined aggregation functions, and control over how the intervals and result labeling are defined. A suite of high performance Cython/C-based resampling functions (including Open-High-Low-Close) have also been implemented.

- Revamp of frequency aliases and support for frequency shortcuts like ‘15min’, or ‘1h30min’

- New DatetimeIndex class supports both fixed frequency and irregular time series. Replaces now deprecated DateRange class

- New PeriodIndex and Period classes for representing time spans and performing calendar logic, including the 12 fiscal quarterly frequencies <timeseries.quarterly>. This is a partial port of, and a substantial enhancement to, elements of the scikits.timeseries code base. Support for conversion between PeriodIndex and DatetimeIndex

- New Timestamp data type subclasses datetime.datetime, providing the same interface while enabling working with nanosecond-resolution data. Also provides easy time zone conversions.

- Enhanced support for time zones. Add tz_convert and tz_localize methods to TimeSeries and DataFrame. All timestamps are stored as UTC; Timestamps from DatetimeIndex objects with time zone set will be localized to local time. Time zone conversions are therefore essentially free. User needs to know very little about pytz library now; only time zone names as strings are required. Time zone-aware timestamps are equal if and only if their UTC timestamps match. Operations between time zone-aware time series with different time zones will result in a UTC-indexed time series.

- Time series string indexing conveniences / shortcuts: slice years, year and month, and index values with strings

- Enhanced time series plotting; adaptation of scikits.timeseries matplotlib-based plotting code

- New date_range, bdate_range, and period_range factory functions

- Robust frequency inference function infer_freq and inferred_freq property of DatetimeIndex, with option to infer frequency on construction of DatetimeIndex

- to_datetime function efficiently parses array of strings to DatetimeIndex. DatetimeIndex will parse array or list of strings to datetime64

- Optimized support for datetime64-dtype data in Series and DataFrame columns

- New NaT (Not-a-Time) type to represent NA in timestamp arrays

- Optimize Series.asof for looking up “as of” values for arrays of timestamps

- Milli, Micro, Nano date offset objects

- Can index time series with datetime.time objects to select all data at particular time of day (TimeSeries.at_time) or between two times (TimeSeries.between_time)

- Add tshift method for leading/lagging using the frequency (if any) of the index, as opposed to a naive lead/lag using shift
Other new features

- New `cut` and `qcut` functions (like R’s `cut` function) for computing a categorical variable from a continuous variable by binning values either into value-based (`cut`) or quantile-based (`qcut`) bins
- Rename `Factor` to `Categorical` and add a number of usability features
- Add `limit` argument to `fillna/reindex`
- More flexible multiple function application in `GroupBy`, and can pass list (name, function) tuples to get result in particular order with given names
- Add flexible `replace` method for efficiently substituting values
- Enhanced `read_csv/read_table` for reading time series data and converting multiple columns to dates
- Add `comments` option to parser functions: `read_csv`, etc.
- Add `dayfirst` option to parser functions for parsing international DD/MM/YYYY dates
- Allow the user to specify the CSV reader `dialect` to control quoting etc.
- Handling `thousands` separators in `read_csv` to improve integer parsing.
- Enable unstacking of multiple levels in one shot. Alleviate `pivot_table` bugs (empty columns being introduced)
- Move to klib-based hash tables for indexing; better performance and less memory usage than Python’s dict
- Add first, last, min, max, and prod optimized `GroupBy` functions
- New `ordered_merge` function
- Add flexible `comparison` instance methods `eq`, `ne`, `lt`, `gt`, etc. to `DataFrame`, `Series`
- Improve `scatter_matrix` plotting function and add histogram or kernel density estimates to diagonal
- Add `kde` plot option for density plots
- Support for converting `DataFrame` to R `data.frame` through rpy2
- Improved support for complex numbers in `Series` and `DataFrame`
- Add `pct_change` method to all data structures
- Add `max_colwidth` configuration option for `DataFrame` console output
- Interpolate `Series` values using index values
- Can select multiple columns from `GroupBy`
- Add `update` methods to `Series/DataFrame` for updating values in place
- Add `any` and `all` method to `DataFrame`

New plotting methods

```python
import pandas as pd

fx = pd.read_pickle("data/fx_prices")
import matplotlib.pyplot as plt

Series.plot now supports a secondary_y option:
```
Vytaitus Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, 'kde' is a new option:

```python
s = pd.Series(np.concatenate((np.random.randn(1000), np.random.randn(1000) * 0.5 + 3)))
plt.figure()
s.hist(density=True, alpha=0.2)
s.plot(kind="kde")
```

See the plotting page for much more.

**Other API changes**

- Deprecation of `offset`, `time_rule`, and `timeRule` arguments names in time series functions. Warnings will be printed until pandas 0.9 or 1.0.

**Potential porting issues for pandas <= 0.7.3 users**

The major change that may affect you in pandas 0.8.0 is that time series indexes use NumPy’s `datetime64` data type instead of `dtype=object` arrays of Python’s built-in `datetime.datetime` objects. `DateRange` has been replaced by `DatetimeIndex` but otherwise behaved identically. But, if you have code that converts `DateRange` or `Index` objects that used to contain `datetime.datetime` values to plain NumPy arrays, you may have bugs lurking with code using scalar values because you are handing control over to NumPy:

```python
In [1]: import datetime
In [2]: rng = pd.date_range("1/1/2000", periods=10)
In [3]: rng[5]
Out[3]: Timestamp('2000-01-06 00:00:00', freq='D')
In [4]: isinstance(rng[5], datetime.datetime)
Out[4]: True
In [5]: rng_asarray = np.asarray(rng)
In [6]: scalar_val = rng_asarray[5]
In [7]: type(scalar_val)
Out[7]: numpy.datetime64
```

pandas’s `Timestamp` object is a subclass of `datetime.datetime` that has nanosecond support (the nanosecond field store the nanosecond value between 0 and 999). It should substitute directly into any code that used `datetime.datetime` values before. Thus, I recommend not casting `DatetimeIndex` to regular NumPy arrays.

If you have code that requires an array of `datetime.datetime` objects, you have a couple of options. First, the `astype(object)` method of `DatetimeIndex` produces an array of `Timestamp` objects:
In [8]: stamp_array = rng.astype(object)

In [9]: stamp_array
Out[9]:
Index([2000-01-01 00:00:00, 2000-01-02 00:00:00, 2000-01-03 00:00:00,
2000-01-04 00:00:00, 2000-01-05 00:00:00, 2000-01-06 00:00:00,
2000-01-07 00:00:00, 2000-01-08 00:00:00, 2000-01-09 00:00:00,
2000-01-10 00:00:00],
dtype='object')

In [10]: stamp_array[5]
Out[10]: Timestamp('2000-01-06 00:00:00', freq='D')

To get an array of proper datetime.datetime objects, use the to_pydatetime method:

In [11]: dt_array = rng.to_pydatetime()

In [12]: dt_array
Out[12]:
array([datetime.datetime(2000, 1, 1, 0, 0),
      datetime.datetime(2000, 1, 2, 0, 0),
      datetime.datetime(2000, 1, 3, 0, 0),
      datetime.datetime(2000, 1, 4, 0, 0),
      datetime.datetime(2000, 1, 5, 0, 0),
      datetime.datetime(2000, 1, 6, 0, 0),
      datetime.datetime(2000, 1, 7, 0, 0),
      datetime.datetime(2000, 1, 8, 0, 0),
      datetime.datetime(2000, 1, 9, 0, 0),
      datetime.datetime(2000, 1, 10, 0, 0)], dtype=object)

In [13]: dt_array[5]
Out[13]: datetime.datetime(2000, 1, 6, 0, 0)

matplotlib knows how to handle datetime.datetime but not Timestamp objects. While I recommend that you plot time series using TimeSeries.plot, you can either use to_pydatetime or register a converter for the Timestamp type. See matplotlib documentation for more on this.

Warning: There are bugs in the user-facing API with the nanosecond datetime64 unit in NumPy 1.6. In particular, the string version of the array shows garbage values, and conversion to dtype=object is similarly broken.

In [14]: rng = pd.date_range("1/1/2000", periods=10)

In [15]: rng
Out[15]:
      '2000-01-09', '2000-01-10'],
dtype='datetime64[ns]', freq='D')

In [16]: np.asarray(rng)
Out[16]:
array(['2000-01-01T00:00:00.000000000', '2000-01-02T00:00:00.000000000',
      '2000-01-03T00:00:00.000000000', '2000-01-04T00:00:00.000000000',
      '2000-01-05T00:00:00.000000000', '2000-01-06T00:00:00.000000000',
      ... '2000-01-10T00:00:00.000000000', '2000-01-11T00:00:00.000000000',
      '2000-01-12T00:00:00.000000000', '2000-01-13T00:00:00.000000000',
      '2000-01-14T00:00:00.000000000', '2000-01-15T00:00:00.000000000',
      '2000-01-16T00:00:00.000000000', '2000-01-17T00:00:00.000000000',
      '2000-01-18T00:00:00.000000000', '2000-01-19T00:00:00.000000000',
      '2000-01-20T00:00:00.000000000', '2000-01-21T00:00:00.000000000',
      '2000-01-22T00:00:00.000000000', '2000-01-23T00:00:00.000000000',
      '2000-01-24T00:00:00.000000000', '2000-01-25T00:00:00.000000000',
      '2000-01-26T00:00:00.000000000', '2000-01-27T00:00:00.000000000',
      '2000-01-28T00:00:00.000000000', '2000-01-29T00:00:00.000000000',
      '2000-01-30T00:00:00.000000000', '2000-01-31T00:00:00.000000000',
      '2000-02-01T00:00:00.000000000'], dtype='datetime64[ns]')
In [17]: converted = np.asarray(rng, dtype=object)

In [18]: converted[5]
Out[18]: Timestamp('2000-01-06 00:00:00', freq='D')

Trust me: don’t panic. If you are using NumPy 1.6 and restrict your interaction with datetime64 values to pandas’s API you will be just fine. There is nothing wrong with the data-type (a 64-bit integer internally); all of the important data processing happens in pandas and is heavily tested. I strongly recommend that you do not work directly with datetime64 arrays in NumPy 1.6 and only use the pandas API.

Support for non-unique indexes: In the latter case, you may have code inside a try:... catch: block that failed due to the index not being unique. In many cases it will no longer fail (some method like append still check for uniqueness unless disabled). However, all is not lost: you can inspect index.is_unique and raise an exception explicitly if it is False or go to a different code branch.

Contributors

A total of 27 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam Klein
- Chang She
- David Zaslavsky +
- Eric Chlebek +
- Jacques Kvam
- Kamil Kisiel
- Kelsey Jordahl +
- Kieran O’Mahony +
- Lorenzo Bolla +
- Luca Beltrame
- Marc Abramowitz +
- Mark Wiebe +
- Paddy Mullen +
- Peng Yu +
- Roy Hyunjin Han +
- RuiDC +
- Senthil Palanisami +
- Skipper Seabold
- Stefan van der Walt +
5.24 Version 0.7

5.24.1 Version 0.7.3 (April 12, 2012)

This is a minor release from 0.7.2 and fixes many minor bugs and adds a number of nice new features. There are also a couple of API changes to note; these should not affect very many users, and we are inclined to call them “bug fixes” even though they do constitute a change in behavior. See the full release notes or issue tracker on GitHub for a complete list.

New features

- New fixed width file reader, read_fwf
- New scatter_matrix function for making a scatter plot matrix

```
from pandas.tools.plotting import scatter_matrix

scatter_matrix(df, alpha=0.2) # noqa F821
```

- Add stacked argument to Series and DataFrame's plot method for stacked bar plots.

```
df.plot(kind="bar", stacked=True) # noqa F821

df.plot(kind="barh", stacked=True) # noqa F821
```

- Add log x and y scaling options to DataFrame.plot and Series.plot
- Add kurt methods to Series and DataFrame for computing kurtosis

NA boolean comparison API change

Reverted some changes to how NA values (represented typically as NaN or None) are handled in non-numeric Series:

```
in [1]: series = pd.Series(['"Steve", np.nan, "Joe"'])

In [2]: series == "Steve"
Out[2]:
0   True
1   False
```

(continues on next page)
In comparisons, NA / NaN will always come through as False except with != which is True. Be very careful with boolean arithmetic, especially negation, in the presence of NA data. You may wish to add an explicit NA filter into boolean array operations if you are worried about this:

```
In [4]: mask = series == "Steve"

In [5]: series[mask & series.notnull()]
```

While propagating NA in comparisons may seem like the right behavior to some users (and you could argue on purely technical grounds that this is the right thing to do), the evaluation was made that propagating NA everywhere, including in numerical arrays, would cause a large amount of problems for users. Thus, a "practicality beats purity" approach was taken. This issue may be revisited at some point in the future.

**Other API changes**

When calling apply on a grouped Series, the return value will also be a Series, to be more consistent with the groupby behavior with DataFrame:

```
In [6]: df = pd.DataFrame(
    ...:     {  
    ...:       "A": ["foo", "bar", "foo", "bar", "foo", "foo", "foo"],  
    ...:       "B": ["one", "one", "two", "three", "two", "two", "one"],  
    ...:       "C": np.random.randn(8),  
    ...:       "D": np.random.randn(8),  
    ...:     })

In [7]: df
```

```
Out[7]:
A   B   C             D
  0  foo one  0.469112   -0.861849
  1  bar one  -0.282863   -2.104569
  2  foo two  -1.509059   -0.494929
  3  bar three  -1.135632    1.071804
  4  foo two   1.212112    0.721555
  5  bar two  -0.173215    -0.706771
  6  foo one  -0.119209   -1.039575
  7  foo three  -1.044236    0.271860
```

(continues on next page)
[8 rows x 4 columns]

In [8]: grouped = df.groupby("A")["C"]

In [9]: grouped.describe()
Out[9]:
          count    mean      std      min     25%      50%     75%      max
A
   bar    3.0 -0.530570  0.526860  -1.135632 -0.709248 -0.282863 -0.228039 -0.173215
   foo    5.0 -0.150572  1.113308  -1.509059 -1.044236  0.119209  0.469112  1.212112

[2 rows x 8 columns]

In [10]: grouped.apply(lambda x: x.sort_values()[-2:])  # top 2 values
Out[10]:
          A
bar   1  -0.282863
      5  -0.173215
foo   0   0.469112
      4   1.212112
Name: C, Length: 4, dtype: float64

Contributors

A total of 15 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Abraham Flaxman +
• Adam Klein
• Andreas H. +
• Chang She
• Dieter Vandenbussche
• Jacques Kvam +
• K.-Michael Aye +
• Kamil Kisiel +
• Martin Blais +
• Skipper Seabold
• Thomas Kluvyer
• Wes McKinney
• Wouter Overmeire
• Yaroslav Halchenko
• lgautier +
5.24.2 Version 0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

New features

- Add additional tie-breaking methods in DataFrame.rank (GH874)
- Add ascending parameter to rank in Series, DataFrame (GH875)
- Add coerce_float option to DataFrame.from_records (GH893)
- Add sort_columns parameter to allow unsorted plots (GH918)
- Enable column access via attributes on GroupBy (GH882)
- Can pass dict of values to DataFrame.fillna (GH661)
- Can select multiple hierarchical groups by passing list of values in .ix (GH134)
- Add axis option to DataFrame.fillna (GH174)
- Add level keyword to drop for dropping values from a level (GH159)

Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)

Contributors

A total of 12 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam Klein
- Benjamin Gross +
- Dan Birken +
- Dieter Vandenbussche
- Josh +
- Thomas Kluyver
- Travis N. Vaught +
- Wes McKinney
- Wouter Overmeire
- claudiobertoldi +
- elpres +
- joshuaar +
5.24.3 Version 0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

New features

- Add `to_clipboard` function to pandas namespace for writing objects to the system clipboard (GH774)
- Add `itertuples` method to DataFrame for iterating through the rows of a dataframe as tuples (GH818)
- Add ability to pass `fill_value` and method to DataFrame and Series align method (GH806, GH807)
- Add `fill_value` option to reindex, align methods (GH784)
- Enable concat to produce DataFrame from Series (GH787)
- Add `between` method to Series (GH802)
- Add HTML representation hook to DataFrame for the IPython HTML notebook (GH773)
- Support for reading Excel 2007 XML documents using openpyxl

Performance improvements

- Improve performance and memory usage of fillna on DataFrame
- Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH787)

Contributors

A total of 9 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam Klein
- Brian Granger +
- Chang She
- Dieter Vandenbussche
- Josh Klein
- Steve +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
5.24.4 Version 0.7.0 (February 9, 2012)

New features

- New unified *merge function* for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains (GH220, GH249, GH267)

- New unified *concatenation function* for concatenating Series, DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of Series.append and DataFrame.append (GH468, GH479, GH273)

- Can pass multiple DataFrames to DataFrame.append to concatenate (stack) and multiple Series to Series.append too

- *Can* pass list of dicts (e.g., a list of JSON objects) to DataFrame constructor (GH526)

- You can now set *multiple columns* in a DataFrame via __getitem__, useful for transformation (GH342)

- Handle differently-indexed output values in DataFrame.apply (GH498)

```python
In [1]: df = pd.DataFrame(np.random.randn(10, 4))
In [2]: df.apply(lambda x: x.describe())
Out[2]:
         0         1         2         3
count 10.000000 10.000000 10.000000 10.000000
mean  0.190912 -0.395125 -0.731920 -0.403130
std   0.730951  0.813266  1.112016  0.961912
min  -0.861849 -2.104569 -1.776904 -1.469388
25%   -0.411391 -0.698728 -1.501401 -1.076610
50%   0.380863  0.057974 -1.191943 -1.004091
75%   0.658444  0.057974 -0.034326  0.461706
max   1.212112  0.577046  1.643563  1.071804
[8 rows x 4 columns]
```

- *Add* reorder_levels method to Series and DataFrame (GH534)

- *Add* dict-like get function to DataFrame and Panel (GH521)

- *Add* DataFrame.iterrows method for efficiently iterating through the rows of a DataFrame

- Add DataFrame.to_panel with code adapted from LongPanel.to_long

- *Add* reindex_axis method added to DataFrame

- *Add* level option to binary arithmetic functions on DataFrame and Series

- *Add* level option to the reindex and align methods on Series and DataFrame for broadcasting values across a level (GH542, GH552, others)

- Add attribute-based item access to Panel and add IPython completion (GH563)

- *Add* logy option to Series.plot for log-scaling on the Y axis

- *Add* index and header options to DataFrame.to_string

- *Can* pass multiple DataFrames to DataFrame.join to join on index (GH115)

- *Can* pass multiple Panels to Panel.join (GH115)

- *Added* justify argument to DataFrame.to_string to allow different alignment of column headers
• Add sort option to GroupBy to allow disabling sorting of the group keys for potential speedups (GH595)
• Can pass MaskedArray to Series constructor (GH563)
• Add Panel item access via attributes and IPython completion (GH554)
• Implement DataFrame.lookup, fancy-indexing analogue for retrieving values given a sequence of row and column labels (GH338)
• Can pass a list of functions to aggregate with groupby on a DataFrame, yielding an aggregated result with hierarchical columns (GH166)
• Can call cummin and cummax on Series and DataFrame to get cumulative minimum and maximum, respectively (GH647)
• value_range added as utility function to get min and max of a dataframe (GH288)
• Added encoding argument to read_csv, read_table, to_csv and from_csv for non-ascii text (GH717)
• Added abs method to pandas objects
• Added crosstab function for easily computing frequency tables
• Added isin method to index objects
• Added level argument to xs method of DataFrame.

API changes to integer indexing

One of the potentially riskiest API changes in 0.7.0, but also one of the most important, was a complete review of how integer indexes are handled with regard to label-based indexing. Here is an example:

```
In [3]: s = pd.Series(np.random.randn(10), index=range(0, 20, 2))
In [4]: s
Out[4]:
0   -1.294524
2    0.413738
4    0.276662
6   -0.472035
8   -0.013960
10  -0.362543
12  -0.006154
14  -0.923061
16   0.895717
18   0.805244
Length: 10, dtype: float64
```

```
In [5]: s[0]
Out[5]: -1.2945235902555294
```

```
In [6]: s[2]
Out[6]: 0.41373810535784006
```

```
In [7]: s[4]
Out[7]: 0.2766617129497566
```

This is all exactly identical to the behavior before. However, if you ask for a key not contained in the Series, in versions 0.6.1 and prior, Series would fall back on a location-based lookup. This now raises a KeyError:
In [2]: s[1]
KeyError: 1

This change also has the same impact on DataFrame:

In [3]: df = pd.DataFrame(np.random.randn(8, 4), index=range(0, 16, 2))

In [4]: df
0   1   2   3
--- --- --- ---
0  0.88427  0.3363 -0.1787  0.03162
2  0.14451 -0.1415  0.2504  0.58374
4 -1.44779 -0.9186 -1.4996  0.27163
6  0.26598 -2.4184 -0.2658  0.11503
8  0.58776  0.3144 -0.8566  0.27163
10 0.10940 -0.7175 -1.0108  0.47990
12 -1.16919 -0.3087 -0.6049 -0.43544
14 0.07337  0.3410  0.0424 -0.16037

In [5]: df.ix[3]
KeyError: 3

In order to support purely integer-based indexing, the following methods have been added:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.iget_value(i)</td>
<td>Retrieve value stored at location i</td>
</tr>
<tr>
<td>Series.iget(i)</td>
<td>Alias for iget_value</td>
</tr>
<tr>
<td>DataFrame.irow(i)</td>
<td>Retrieve the i-th row</td>
</tr>
<tr>
<td>DataFrame.icol(j)</td>
<td>Retrieve the j-th column</td>
</tr>
<tr>
<td>DataFrame.iget_value(i, j)</td>
<td>Retrieve the value at row i and column j</td>
</tr>
</tbody>
</table>

### API tweaks regarding label-based slicing

Label-based slicing using ix now requires that the index be sorted (monotonic) unless both the start and endpoint are contained in the index:

In [1]: s = pd.Series(np.random.randn(6), index=list('gmkaec'))

In [2]: s
Out[2]:
        g   -1.182230
        m   -0.276183
        k   -0.243550
        a   1.628992
        e   0.073308
        c  -0.539890
dtype: float64

Then this is OK:

In [3]: s.ix['k':'e']
Out[3]:
        k   -0.243550
(continues on next page)
But this is not:

```
In [12]: s.ix['b':'h']
KeyError 'b'
```

If the index had been sorted, the “range selection” would have been possible:

```
In [4]: s2 = s.sort_index()
In [5]: s2
Out[5]:
a  1.628992
b  0.73308
c -0.539890
e  0.073308
g -1.182230
k -0.243550
m -0.276183
```
```
In [6]: s2.ix['b':'h']
Out[6]:
c -0.539890
e  0.073308
g -1.182230
```

### Changes to Series [] operator

As as notational convenience, you can pass a sequence of labels or a label slice to a Series when getting and setting values via [] (i.e. the __getitem__ and __setitem__ methods). The behavior will be the same as passing similar input to ix except in the case of integer indexing:

```
In [8]: s = pd.Series(np.random.randn(6), index=list('acegkm'))

In [9]: s
Out[9]:
a -1.206412
c  2.565646
e  1.431256
g  1.340309
k -1.170299
m -0.226169
Length: 6, dtype: float64

In [10]: s[['m', 'a', 'c', 'e']]
Out[10]:
m -0.226169

(continues on next page)
a -1.206412
c 2.565646
e 1.431256
Length: 4, dtype: float64

In [11]: s['b':'l']
Out[11]:
c 2.565646
e 1.431256
g 1.340309
k -1.170299
Length: 4, dtype: float64

In [12]: s['c':'k']
Out[12]:
c 2.565646
e 1.431256
g 1.340309
k -1.170299
Length: 4, dtype: float64

In the case of integer indexes, the behavior will be exactly as before (shadowing ndarray):

In [13]: s = pd.Series(np.random.randn(6), index=range(0, 12, 2))

In [14]: s[[4, 0, 2]]
Out[14]:
4 0.132003
0 0.410835
2 0.813850
Length: 3, dtype: float64

In [15]: s[1:5]
Out[15]:
2 0.813850
4 0.132003
6 -0.827317
8 -0.076467
Length: 4, dtype: float64

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use ix.

Other API changes

- The deprecated LongPanel class has been completely removed
- If Series.sort is called on a column of a DataFrame, an exception will now be raised. Before it was possible to accidentally mutate a DataFrame's column by doing df[col].sort() instead of the side-effect free method df[col].order() (GH316)
- Miscellaneous renames and deprecations which will (harmlessly) raise FutureWarning
- drop added as an optional parameter to DataFrame.reset_index (GH699)
Performance improvements

- **Cythonized GroupBy aggregations** no longer presort the data, thus achieving a significant speedup (GH93). GroupBy aggregations with Python functions significantly sped up by clever manipulation of the ndarray data type in Cython (GH496).
- Better error message in DataFrame constructor when passed column labels don’t match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don’t print length by default in Series.to_string, add length option (GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of Series._getitem__ for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in setup.py if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also (GH536)
- Default name assignment when calling reset_index on DataFrame with a regular (non-hierarchical) index (GH476)
- Use Cythonized groupers when possible in Series/DataFrame stat ops with level parameter passed (GH545)
- Ported skiplist data structure to C to speed up rolling_median by about 5-10x in most typical use cases (GH374)

Contributors

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Adam Klein
- Bayle Shanks +
- Chris Billington +
- Dieter Vandenbussche
- Fabrizio Pollastri +
- Graham Taylor +
- Gregg Lind +
- Josh Klein +
- Luca Beltrame
- Olivier Grisel +
- Skipper Seabold
- Thomas Kluyver
5.25 Version 0.6

5.25.1 Version 0.6.1 (December 13, 2011)

New features

• Can append single rows (as Series) to a DataFrame
• Add Spearman and Kendall rank correlation options to Series.corr and DataFrame.corr (GH428)
• Added get_value and set_value methods to Series, DataFrame, and Panel for very low-overhead access (>2x faster in many cases) to scalar elements (GH437, GH438). set_value is capable of producing an enlarged object.
• Add PyQt table widget to sandbox (GH435)
• DataFrame.align can accept Series arguments and an axis option (GH461)
• Implement new SparseArray and SparseList data structures. SparseSeries now derives from SparseArray (GH463)
• Better console printing options (GH453)
• Implement fast data ranking for Series and DataFrame, fast versions of scipy.stats.rankdata (GH428)
• Implement DataFrame.from_items alternate constructor (GH444)
• DataFrame.convert_objects method for inferring better dtypes for object columns (GH302)
• Add rolling_corr_pairwise function for computing Panel of correlation matrices (GH189)
• Add margins option to pivot_table for computing subgroup aggregates (GH114)
• Add Series.from_csv function (GH482)
• Can pass DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH #462)
• MultiIndex.get_level_values can accept the level name
Performance improvements

- Improve memory usage of DataFrame.describe (do not copy data unnecessarily) (PR #425)
- Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
- Fix performance regression in cross-sectional count in DataFrame, affecting DataFrame.dropna speed
- Column deletion in DataFrame copies no data (computes views on blocks) (GH #158)

Contributors

A total of 7 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Dieter Vandenbussche
- Fernando Perez +
- Jev Kuznetsov +
- Joon Ro
- Ralph Bean +
- Wes McKinney
- Wouter Overmeire

5.25.2 Version 0.6.0 (November 25, 2011)

New features

- Added melt function to pandas.core.reshape
- Added level parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Added head and tail methods to Series, analogous to DataFrame (GH296)
- Added Series.isin function which checks if each value is contained in a passed sequence (GH289)
- Added float_format option to Series.to_string
- Added skip_footer (GH291) and converters (GH343) options to read_csv and read_table
- Added drop_duplicates and duplicated functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- Implemented operators ‘&’, ‘|’, ‘^’, ‘-’ on DataFrame (GH347)
- Added Series.mad, mean absolute deviation
- Added QuarterEnd DateOffset (GH321)
- Added dot to DataFrame (GH65)
- Added orient option to Panel.from_dict (GH359, GH301)
- Added orient option to DataFrame.from_dict
- Added passing list of tuples or list of lists to DataFrame.from_records (GH357)
- Added multiple levels to groupby (GH103)
- Allow multiple columns in by argument of DataFrame.sort_index (GH92, GH362)
• *Added* fast get_value and put_value methods to DataFrame (GH360)
• *Added* cov instance methods to Series and DataFrame (GH194, GH362)
• *Added* kind= 'bar' option to DataFrame.plot (GH348)
• *Added* idxmin and idxmax to Series and DataFrame (GH286)
• *Added* read_clipboard function to parse DataFrame from clipboard (GH300)
• *Added* nunique function to Series for counting unique elements (GH297)
• *Made* DataFrame constructor use Series name if no columns passed (GH373)
• *Support* regular expressions in read_table/read_csv (GH364)
• *Added* DataFrame.to_html for writing DataFrame to HTML (GH387)
• *Added* support for MaskedArray data in DataFrame, masked values converted to NaN (GH396)
• *Added* DataFrame.boxplot function (GH368)
• *Can* pass extra args, kwds to DataFrame.apply (GH376)
• *Implement* DataFrame.join with vector on argument (GH312)
• *Added* legend boolean flag to DataFrame.plot (GH324)
• *Can* pass multiple levels to stack and unstack (GH370)
• *Can* pass multiple values columns to pivot_table (GH381)
• *Use* Series name in GroupBy for result index (GH363)
• *Added* raw option to DataFrame.apply for performance if only need ndarray (GH309)
• *Added* proper, tested weighted least squares to standard and panel OLS (GH303)

**Performance enhancements**

• VBENCH Cythonized cache_readonly, resulting in substantial micro-performance enhancements throughout the code base (GH361)
• VBENCH Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than np.apply_along_axis (GH309)
• VBENCH Improved performance of MultiIndex.from_tuples
• VBENCH Special Cython matrix iterator for applying arbitrary reduction operations
• VBENCH + DOCUMENT Add raw option to DataFrame.apply for getting better performance when
• VBENCH Faster cythonized count by level in Series and DataFrame (GH341)
• VBENCH? Significant GroupBy performance enhancement with multiple keys with many “empty” combinations
• VBENCH New Cython vectorized function map_infer speeds up Series.apply and Series.map significantly when passed elementwise Python function, motivated by (GH355)
• VBENCH Significantly improved performance of Series.order, which also makes np.unique called on a Series faster (GH327)
• VBENCH Vastly improved performance of GroupBy on axes with a MultiIndex (GH299)
Contributors

A total of 8 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Adam Klein +
• Chang She +
• Dieter Vandenbussche
• Jeff Hammerbacher +
• Nathan Pinger +
• Thomas Kluyver
• Wes McKinney
• Wouter Overmeire +

5.26 Version 0.5

5.26.1 Version 0.5.0 (October 24, 2011)

New features

• Added DataFrame.align method with standard join options
• Added parse_dates option to read_csv and read_table methods to optionally try to parse dates in the index columns
• Added nrows, chunksize, and iterator arguments to read_csv and read_table. The last two return a new TextParser class capable of lazily iterating through chunks of a flat file (GH242)
• Added ability to join on multiple columns in DataFrame.join (GH214)
• Added private _get_duplicates function to Index for identifying duplicate values more easily (ENH5c)
• Added column attribute access to DataFrame.
• Added Python tab completion hook for DataFrame columns. (GH233, GH230)
• Implemented Series.describe for Series containing objects (GH241)
• Added inner join option to DataFrame.join when joining on key(s) (GH248)
• Implemented selecting DataFrame columns by passing a list to __getitem__ (GH253)
• Implemented & and | to intersect / union Index objects, respectively (GH261)
• Added pivot_table convenience function to pandas namespace (GH234)
• Implemented Panel.rename_axis function (GH243)
• DataFrame will show index level names in console output (GH334)
• Implemented Panel.take
• Added set_eng_float_format for alternate DataFrame floating point string formatting (ENH61)
• Added convenience set_index function for creating a DataFrame index from its existing columns
• Implemented groupby hierarchical index level name (GH223)
Added support for different delimiters in DataFrame.to_csv (GH244)

Performance enhancements

- VBENCH Major performance improvements in file parsing functions read_csv and read_table
- VBENCH Added Cython function for converting tuples to ndarray very fast. Speeds up many MultiIndex-related operations
- VBENCH Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance (GH211)
- VBENCH Improved speed of DataFrame.xs on mixed-type DataFrame objects by about 5x, regression from 0.3.0 (GH215)
- VBENCH With new DataFrame.align method, speeding up binary operations between differently-indexed DataFrame objects by 10-25%.
- VBENCH Significantly sped up conversion of nested dict into DataFrame (GH212)
- VBENCH Significantly speed up DataFrame.__repr__ and count on large mixed-type DataFrame objects

Contributors

A total of 9 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

- Aman Thakral +
- Luca Beltrame +
- Nick Pentreath +
- Skipper Seabold
- Thomas Kluyver +
- Wes McKinney
- Yaroslav Halchenko +
- lodagro +
- unknown +

5.27 Version 0.4

5.27.1 Versions 0.4.1 through 0.4.3 (September 25 - October 9, 2011)

New features

- Added Python 3 support using 2to3 (GH200)
- Added name attribute to Series, now prints as part of Series.__repr__
- Added instance methods isnull and notnull to Series (GH209, GH203)
- Added Series.align method for aligning two series with choice of join method (ENH56)
- Added method get_level_values to MultiIndex (GH188)
• Set values in mixed-type DataFrame objects via .ix indexing attribute (GH135)
• Added new DataFrame methods get_dtype_counts and property dtypes (ENHdc)
• Added ignore_index option to DataFrame.append to stack DataFrames (ENH1b)
• read_csv tries to sniff delimiters using csv.Sniffer (GH146)
• read_csv can read multiple columns into a MultiIndex; DataFrame's to_csv method writes out a corresponding MultiIndex (GH151)
• DataFrame.rename has a new copy parameter to rename a DataFrame in place (ENHed)
• Enable unstacking by name (GH142)
• Enable sortlevel to work by level (GH141)

Performance enhancements

• Altered binary operations on differently-indexed SparseSeries objects to use the integer-based (dense) alignment logic which is faster with a larger number of blocks (GH205)
• Wrote faster Cython data alignment / merging routines resulting in substantial speed increases
• Improved performance of isnull and notnull, a regression from v0.3.0 (GH187)
• Refactored code related to DataFrame.join so that intermediate aligned copies of the data in each DataFrame argument do not need to be created. Substantial performance increases result (GH176)
• Substantially improved performance of generic Index.intersection and Index.union
• Implemented BlockManager.take resulting in significantly faster take performance on mixed-type DataFrame objects (GH104)
• Improved performance of Series.sort_index
• Significant groupby performance enhancement: removed unnecessary integrity checks in DataFrame internals that were slowing down slicing operations to retrieve groups
• Optimized _ensure_index function resulting in performance savings in type-checking Index objects
• Wrote fast time series merging / joining methods in Cython. Will be integrated later into DataFrame.join and related functions

Contributors

A total of 2 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

• Thomas Kluyver +
• Wes McKinney
[1] https://docs.sqlalchemy.org
[1] https://docs.sqlalchemy.org
pandas, 1